VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Analytické programování v platformě nezávislém jazyce

Analytic Programming Based on a Platform Independent Language

2014 Bc. Tomáš Tyleček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Tomáš Tyleček**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Analytické programování v platformě nezávislém jazyce
Analytic Programming Based on a Platform Independent Language

Zásady pro vypracování:

Cílem práce je vytvoření analytického programování v jazyce, jehož použití je nezávislé na používané výpočetní platformě. V rámci laboratoře oboru navrhněte praktickou ukázku využívající vytvořený algoritmus (např. optimalizovaný matematický model, fitování dat z oblasti astroinformatiky nebo naučenou neuronovou síť). Charakter práce spadá do oblasti programování v oblasti pokročilých evolučních technik.

1. Seznámení se s problematikou analytického programování.
2. Vytvoření algoritmu analytického programování ve vybraném jazyce (Java, ...).
3. Provedení testování na vybraných problémech.
4. Závěr.

Seznam doporučené odborné literatury:

[1] Koza J.R. 1998, Genetic Programming, MIT Press, ISBN 0-262-11189-6, 1998
[2] Koza J.R.,Bennet F.H., Andre D., Keane M. 1999, Genetic Programming III, Morgan Kaufnamm pub., ISBN 1-55860-543-6, 1999
[3] Lampinen Jouni, Zelinka, Ivan, New Ideas in Optimization & Mechanical Engineering Design Optimization by Differential Evolution. Volume 1. London: McGraw-Hill, 1999. 20 p. ISBN 007-709506-5
[4] Kvasnička V., Pospíchal J., Tiňo P., Evolučné algoritmy, STU Bratislava, ISBN 85-246-2000, 2000
[5] Zelinka I.: Analytic Programming by Means of Soma Algorithm. ICICIS'02, First International Conference on Intelligent Computing and Information Systems, Egypt, Cairo, 2002
[6] Zelinka Ivan, Evoluční výpočetní techniky - principy a alikace, BEN, Praha, 2008

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **prof. Ing. Ivan Zelinka, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2014

Tomáš Zlesák

Rád bych na tomto místě poděkoval všem, kteří mě během studia podporovali. Dále děkuji vedoucímu diplomové práce prof. Ing. Ivanu Zelinkovi Ph.D. za vedení této práce.

Poděkování

Tato práce byla vypracována s podporou projektu Rozvoj lidských zdrojů ve výzkumu a vývoji moderních soft computingových metod a jejich praktického využití, reg. č. CZ.1.07/2.3.00/20.0072 podpořeného Operačním programem Vzdělávání pro konkurenceschopnost, financovaného ze strukturálních fondů EU a státního rozpočtu ČR.

Abstrakt

Diplomová práce pojednává o experimentální evoluční metodě s názvem Analytické programování a její implementaci v platformě nezávislém jazyce. Dále se práce zabývá testováním této implementace Analytického programování na vybraném problému, čímž je prokládání dat funkcí, kde cílem je získat předpis funkce, která co nejlépe prochází vstupními daty.

Klíčová slova: Analytické programování, Evoluční algoritmy, Evoluční výpočetní techniky, Java, Optimalizace, Aproximace funkcí

Abstract

Diploma thesis deals with experimental evolution method called Analytic programming and it's implementation in platform independent language. The next goal of this work is testing of it's implementation of Analytic programming on selected problem which is data fitting where the goal is to get definition of fuction based on input data.

Keywords: Analytic programming, Evolutionary algorithm, Evolutionary computation techniques, Java, Optimalization, Approximation of functions

Seznam použitých zkratek a symbolů

AP	–	Analytické Programování
EA	–	Evoluční Algoritmy
EVT	–	Evoluční výpočetní techniky
GFS	–	General Functional Set
GUI	–	Graphical User Interface
IDE	–	Integrated Development Environment
DE	–	Diferencialní evoluce
DP	–	Diplomová práce
CV	–	Cost value
JVM	–	Java Virtual Machine
API	–	Application Programming Interface
CR	–	Crossover Ratio
ES	–	Evoluční strategie
PS	–	Particle Swarm
SA	–	Simulated Annealing
SOMA	–	Samo Organizující se Migrační Algoritmus
JSON	–	JavaScript Object Notation
OS	–	Operační systém

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V poslední době je potřeba hledat čím dál tím víc nové optimalizační metody, které by ulehčily optimalizační úlohy, které inženýři provádějí. Se vzrůstající obtížností komplexity úkolů roste i výpočetní čas a zapojení řešitele do procesu. Z toho důvodu je potřeba používat efektivní techniky, které se tyto rizika snaží eliminovat.

Tato *DP* se zabývá právě jednou z těchto technik. Jedná se o experimentální evoluční techniku s názvem Analytické programování. Cílem této práce je provést implementaci této techniky a ověřit její funkčnost na vybraném problému. Tento problém představuje prokládání dat funkcí, kde cílem je získat předpis funkce, která co nejlépe prochází vstupními daty.

Práce je rozdělena do sedmi kapitol, kde první je úvod a poslední je závěr. Dále práce v příloze obsahuje tabulky, kde s jejich pomocí je celý implementovaný program nakonfigurován. Součástí přílohy je také ukázka, jak může vypadat předpis nalezené funkce.

Druhá kapitola, „Úvod do EVT“, si klade za úkol, seznámit čtenáře s problematikou evolučních výpočetních technik (*EVT*). Kapitola se zabývá podstatou evolučních technik, včetně jejich historie a základních charakteristik. Je zde také představen pojem Symbolické regrese, který je úzce spjat s Analytickým programováním (*AP*).

Na tuto kapitolu navazuje kapitola s názvem „Analytické programování“. V této kapitole je teoreticky vysvětleno, co to *AP* je a jaký je princip této techniky. Jsou zde také popsány metody, díky kterým lze výkon *AP* zefektivnit.

Další kapitola se zabývá teoretickým výkladem evolučních algoritmů (*EA*), které byly v rámci implementace *AP* použity. Kapitola nese název „Použité evoluční algoritmy“ a čtenář se zde seznámí s celkem pěti *EA* včetně jejich možných verzí.

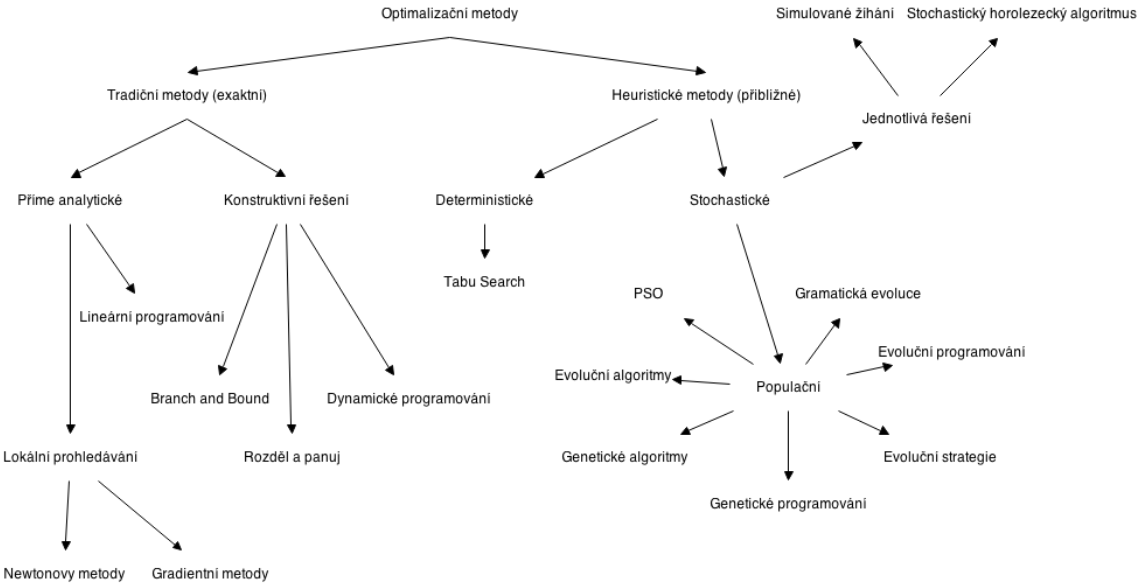
Pátá kapitola s názvem „Implementace“ se zabývá kódem programu. Je zde ukázáno, v čem a jak byla implementace programu provedena a čtenář by po přečtení této kapitoly měl být se strukturou s funkčností programu seznámen. Tato kapitola zároveň slouží jako dokumentace kódu.

Šestá kapitola se zabývá experimenty nad tímto implementovaným programem, kde byly otestovány různé algoritmy v rámci *AP* a zobrazeny jejich výsledky.

2 Úvod do *EVT*

EA jsou velice výkonným nástrojem pro řešení mnoha úloh inženýrské praxe. Často se používají v situacích, kdy řešení dané úlohy nelze efektivně řešit analyticky. Tyto algoritmy můžou být na tyto úlohy aplikovány tak, že dokonce není potřeba často do jejich činnosti zasahovat. V dnešní době vysokých výpočetních výkonů evolučním algoritmům vyloženě nahrává fakt, že řešené úlohy lze definovat jako optimalizační problém, který jsou tyto algoritmy schopny efektivně řešit. Tento problém se převede na matematickou úlohu danou funkčním předpisem a pomocí optimalizace jejich argumentů se nalezne řešení. Mezi příklady aplikace těchto algoritmů v reálném světě patří například optimalizace tloušťky stěny, optimalizace spotřeby letadla pro ušetření prostředků nebo optimalizace nastavení regulátoru. Tyto algoritmy často vyřeší problémy tak elegantně, že jejich používání je v dnešní době velice populární a oblíbené. Tomu nasvědčuje i fakt, že jejich implementace není moc složitá. Pro jejich použití je však potřeba, aby řešitel dané úlohy měl hlubokou znalost řešené problematiky, protože musí správně převést řešený problém na funkci, která bude optimalizována. Tyto algoritmy jsou však často klasickými rigorózními matematiky přehlíženy, protože jejich výsledek často nelze předvídat, takže se matematické důkazy pro tyto algoritmy sestavují velice těžce. [3]

Obecně lze *EA* zařadit do skupiny heuristických algoritmů. Heuristické algoritmy lze dále rozdělit na dvě skupiny. Tou první skupinou jsou algoritmy deterministického typu, kdy v každém jejich kroku je předem známo, co bude následovat, tedy na základě vstupu lze s jistotou předvídat výstup. Druhou skupinou jsou algoritmy založené na náhodě, kterým se říká stochastické. U těchto algoritmů nelze s jistotou určit, jaký bude výstup na základě předloženého vstupu. Dva různé běhy stochastického algoritmu můžou na stejný vstup vytvořit dva úplně různé výstupy. Stochastické algoritmy lze dále dělit na základě toho, zda pracují s populací jedinců nebo pouze s jedním bodem. Základem algoritmů pracujících pouze s jedním bodem je operace sousedství aktuálního řešení, na základě něhož se hledá lepší řešení. Toto rozdělení algoritmů do jednotlivých skupin je zobrazeno na obrázku 1. [4]



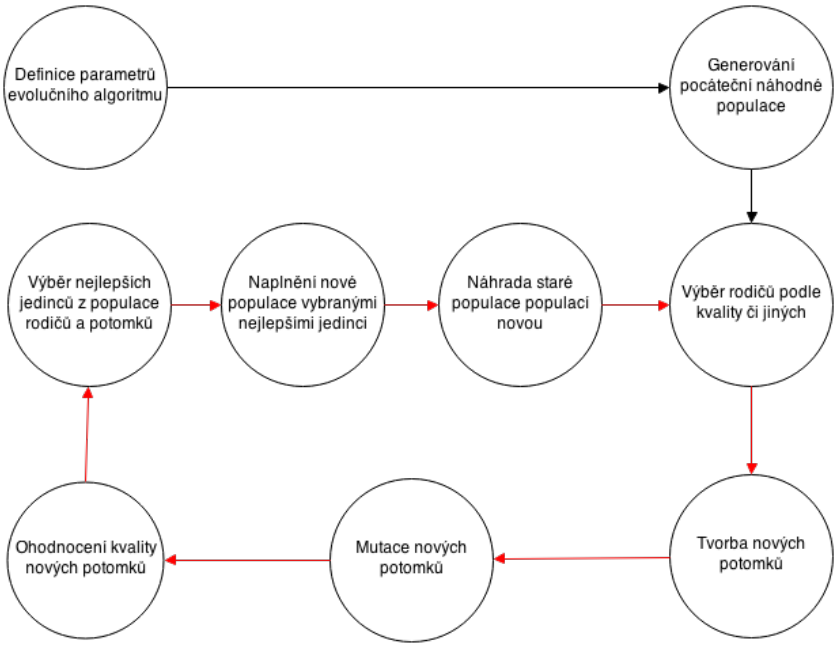
Obrázek 1: Rozdělení optimalizačních algoritmů do skupin

2.1 Historie

Historie vzniku *EA* se datuje ke dvěma nezávislým milníkům, které lze oba považovat za začátek historie *EA*. První z nich nastal v polovině 70. let, kdy byly představeny Johnem Hollandem genetické algoritmy. Druhý z nich se datuje do poloviny 60. let, kdy byly poprvé s úspěchem použity evoluční strategie. S ohledem do hlubší historie, kdy ještě nebylo dostatek výpočetního výkonu, lze tvrdit, že však duchovními otci *EVT* jsou A. M. Turing, N. A. Barricelli a další. V této době totiž formulovali principy, které se podobají dnešním *EA*. Nicméně nejspíše díky nedostatku výpočetního výkonu je nebyli schopni realizovat. [5]

2.2 Evoluční cyklus

EA jsou inspirovány Darwinovou a Mendelovou teorií evoluce. Vycházejí z toho, že rodiče z jednotlivých živočišných druhů plodí své potomky. Tito potomci podléhají mutacím jejich genetického kódu, které můžou být tak zásadní, že nepřežijí, v jejich typickém životním prostředí. Ti co nepřežijí, uvolňují místo novým lepším potomkům. Celý tento proces vymírání a tvorby nových jedinců se po generacích opakuje. Jsou tedy založeny na předávání genomu rodiče na své potomky a uvolňování životního prostoru přeživším jedincům, kteří cyklicky mají potomky. Tento proces vzniku a zániku živočišných druhů, tedy evoluční cyklus, je ve zjednodušené podobě přenesen do počítače způsobem, který je zobrazen na obrázku 2. [4]



Obrázek 2: Evoluční cyklus

Celý proces začíná definováním parametrů *EA*. Tyto parametry řídí běh algoritmu a určují, zda má nebo nemá být algoritmus ukončen. Často jsou v parametrech nastaveny ukončovací kritéria jako například počet generací nebo požadovaná kvalita jedince. Ta je stanovena pomocí účelové funkce, kterou je potřeba taktéž definovat v parametrech *EA*. Tato funkce představuje matematický model problému, který algoritmus řeší. Nalezení maxima či minima této funkce představuje řešení tohoto problému. Můžeme říct, že v kontextu *EA* je účelová funkce ekvivalentem k životnímu prostředí, ve kterém jedinec žije. Proces pokračuje vygenerováním počáteční populace. Tu si lze představit jako matici o M řádcích a N sloupcích, kde N je počet parametrů jedince a M je počet jedinců v populaci. Populace je tedy vygenerována na základě počtu optimalizovaných argumentů účelové funkce a na kriteriích definovaných uživatelem při definování parametrů *EA*. U počáteční populace jsou všechny hodnoty v tabulce nastaveny náhodně. Jednomu řádku tabulky se říká jedinec a představuje jedno konkrétní řešení řešeného problému. Následně jsou všichni jedinci z počáteční populace ohodnoceni pomocí účelové funkce, která byla definována jako parametr algoritmu. Do každého z jedinců se uloží hodnota vrácená účelovou funkcí. Na základě této hodnoty lze následně porovnávat jejich kvalitu. Nyní nastává výběr rodičů na základě jejich kvality. Z těchto vybraných jedinců se pomocí procesu křížení tvoří noví jedinci. Způsob, jakým bude proces křížení proveden, závisí na použitém *EA*. Každý z nich k této problematice přistupuje trochu jiným způsobem. Takto získaní noví jedinci se následně zmutují. Tuto činnost si lze představit tak, že jsou některé jejich čísla náhodně pozměněny. Jedná se o ekvivalenci s biologickou mutací genů. Proces mutace také závisí na použitém *EA*. Každý takhle zmutovaný jedinec se znova ohodnotí pomocí účelové funkce. Z nich se vyberou nejlepší jedinci, kteří zaplní

novou populaci. Stará populace následně umírá a je nahrazena novou populací vzniklou z nejlepších vybraných jedinců po mutaci. Následně se pokračuje opět výběrem rodičů pro tvorbu další populace. Tento postup se opakuje, dokud není dosažen určitý počet cyklů, nebo dokud není nalezeno řešení požadované kvality, tedy alespoň jeden z ukončovacích parametrů je splněn. Tento evoluční cyklus je obecný a pro konkrétní *EA* se může mírně lišit. [6]

2.3 Populace

Pro *EA* je charakteristické, že jsou často založeny na práci s populací jedinců. Tuto populaci si lze představit jako tabulku o *N* řádcích a *M* sloupcích (tab. 1). Sloupce představují jednotlivé jedince, kde každý takový sloupec představuje potencionální řešení problému. Řádky představují argumenty účelové funkce, kde jejich správná kombinace je algoritmy hledána. Kromě těchto argumentů je s jedincem spojena jeho hodnota účelové funkce. V tabulce by byla reprezentována dalším řádkem. Tato hodnota říká, jak je jedinec kvalitní pro další vývoj populace. Při evoluci se však tato hodnota neúčastní vlastního evolučního cyklu, slouží pouze pro uložení informace o kvalitě jedince. [6]

	$\mathbf{J_1}$	$\mathbf{J_2}$	$\mathbf{J_3}$	...	...	$\mathbf{J_m}$
Vhodnost	$50,6$	$62,1$	$5,3$	...	...	$0,21$
$\mathbf{P_1}$	$-12,3$	$4,8$	$0,05$	...	...	J_m
$\mathbf{P_2}$	$3,5$	$11,2$	$-70.,$	...	...	J_m
...	...	...	...	...	...	...
$\mathbf{P_n}$	$158,14$	$-2,1$	$-80,9$	...	...	J_m

Tabulka 1: Náhodná populace jedinců

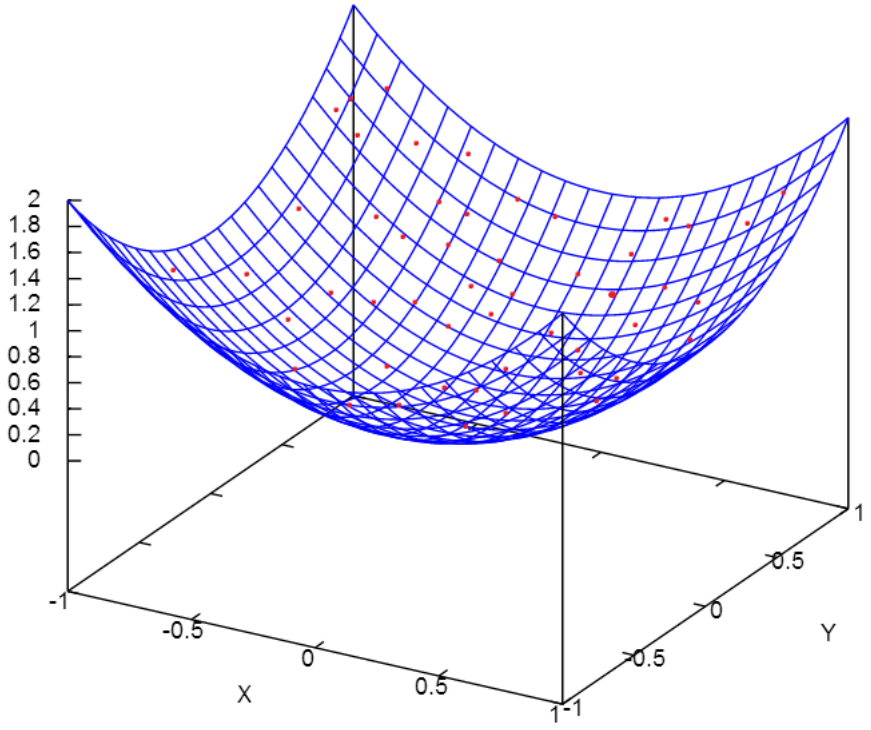
Vytvoření populace probíhá za asistence tzv. specimenu. Jedná se o vzor jedince, na základě kterého se generuje celá počáteční populace. Podle tohoto vzoru pak lze taktéž korigovat parametry jedinců, kteří překročí hranice prohledávaného prostoru. Specimen je definován vztahem 1. [6]

$$Specimen \quad = \{\{Real, \{Lo, Hi\}\}, \{Integer, \{Lo, Hi\}\}, ..., \{Real, \{Lo, Hi\}\}\} \tag{1}$$

Specimen obsahuje pro každý parametr jedince tři konstanty. První specifikuje typ proměnné, zbylé dva definují povolený rozsah hodnoty a to tím způsobem, že druhý parametr představuje dolní hranici povoleného rozsahu a třetí parametr představuje horní hranici povoleného rozsahu. Spolu tedy tvoří interval povolených hodnot. Správná volba těchto hranic je velice důležitý úkol. Při jejich nesprávném zvolení se totiž může stát, že výsledné řešení evoluce nebude možné fyzikálně realizovat, protože například nelze vytvořit stěnu o záporné tloušťce. Na základě vzorového jedince lze populaci vygenerovat pomocí vztahu (2). Díky tomuto vztahu je zajištěno, že všechny parametry jedinců budou ležet v povolených hranicích definovaných tímto vzorem. [6]

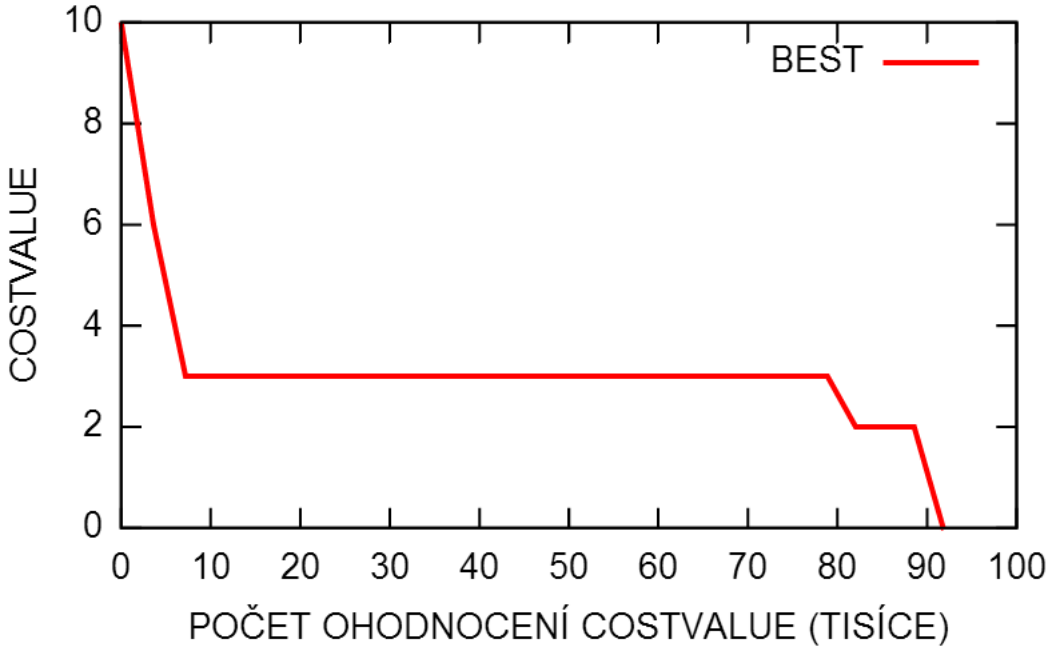
$$P_{i,j} = x_{i,j}^{(0)} = rand(0,1) * (x_{i,j}^{(Hi)} - x_{i,j}^{(Lo)}) + x_{i,j}^{(Lo)} i = 1, ..., M j = 1, ..., N \tag{2}$$

Na obrázku 3 je vizualizovaná náhodně vygenerovaná populace. Jedná se o populaci o padesáti jedincích, kde jedinec obsahuje dva parametry. Účelová funkce je zde vykreslena modře a jednotlivé jedince představují červené body na této funkci. Vizualizace je provedena způsobem, že první parametr jedince představuje proměnnou X a druhý parametr proměnnou Y. Hodnota účelové funkce jedince poté říká, jak vysoko bude jedinec vykreslen. Jedinec je tedy reprezentován v prostoru bodem. Z obrázku lze vyčíst, že jedinci jsou opravdu náhodně umístěni v prostoru možných řešení. Při běhu evoluce se pak tito jedinci pomalu shromažďují okolo jednoho nebo více z extrémů.



Obrázek 3: Náhodná počáteční populace

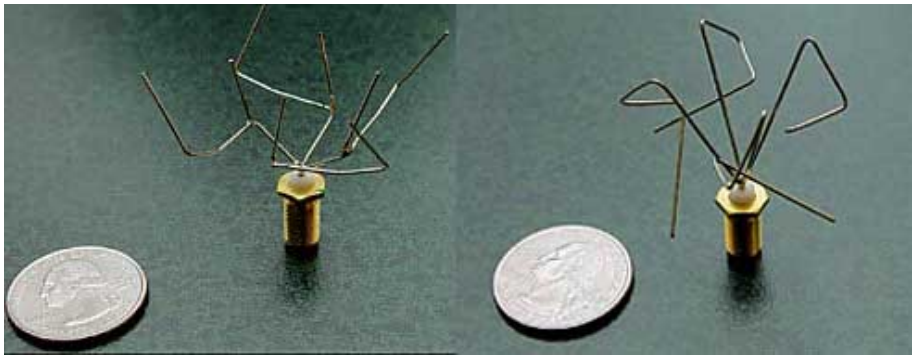
Pro zobrazení postupu celé evoluce se využívá graf závislosti hodnoty účelové funkce na aktuálním počtu jejich ohodnocení. Tento způsob zobrazení totiž bere v potaz fakt, že různé algoritmy během svého cyklu provádí různý počet ohodnocení účelové funkce. Díky tomu je pak možné objektivně porovnávat různé typy algoritmů nezávisle na jejich vnitřní implementaci. Ukázka, jak takový graf může vypadat, je zobrazena na obrázku 4. [4]



Obrázek 4: Graf závislosti hodnoty účelové funkce na aktuálním počtu jejich ohodnocení

2.4 Symbolická regrese

Symbolická regrese je proces, kdy se z malých stavebních kamenů staví složitější struktura, která má popisovat požadované chování. Příkladem může být prokládání zdrojových dat vhodnou funkci, kde tyto zdrojové data jsou chápány jako požadované chování a vhodná funkce jako složitější struktura složená z malých stavebních kamenů. Symbolická regrese může být také použita na vyřešení problémů v reálném světě, jako je například návrh logického obvodu či antény, která odpovídá požadovaným parametrům. Existuje tedy celá řada možných problémů, na které je symbolickou regresi možné aplikovat. Pokud uvažujeme symbolickou regresi v rámci *EA*, tak výhodou tohoto přístupu při hledání řešení problému je fakt, že řešení, které je výstupem symbolické regrese, by člověka často nenapadlo. Tento přístup tedy může objevit nové implementace zadaného problému. Na obrázku 5 je zobrazen příklad, kde byla symbolická regrese využita. Jsou zde zobrazeny antény, které i přes svůj zvláštní tvar, perfektně splňují požadované parametry. [4, 7]



Obrázek 5: Antény vytvořené pomocí symbolické regrese. Obrázek převzat z [1]

V rámci evolučních technik v dnešní době existují tři metody symbolické regrese. Tou nejstarší je genetické programování, které bylo představeno již na konci 80. let minulého století. Tato metoda je rozšířením genetického evolučního algoritmu a je s ním tedy úzce spjata. Od genetického programování se hlavně liší v reprezentaci jedinců. Ti jsou zde reprezentování pomocí symbolických objektů, například stromové struktury. [8, 9]

Druhou metodou je gramatická evoluce, která může být taktéž chápána jako typ genetického programování. Její historie sahá až do roku 1998. Tato metoda je založena na gramatice a na rozdíl od genetického programování je obecnější, protože je navržena tak, aby byla použitelná v libovolném jazyce. Tato vlastnost se odráží i v reprezentaci jedinců, protože jejich reprezentace je binární. [10]

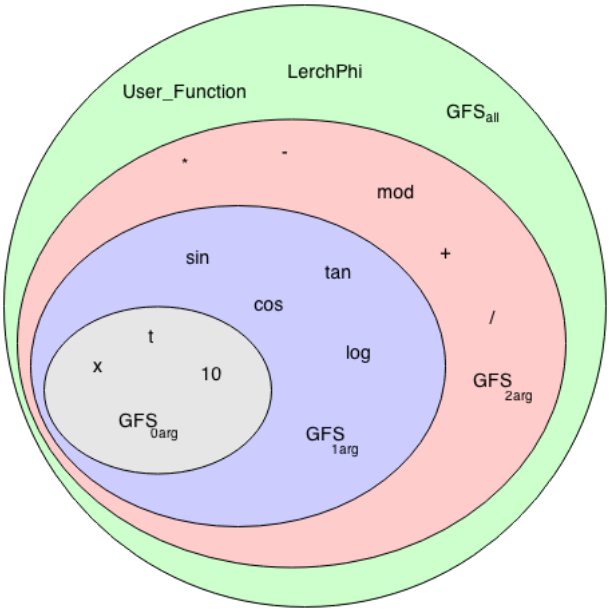
Tato práce se však blíže zabývá pouze poslední metodou symbolické regrese a tou je Analytické programování, které je z těchto jmenovaných metod nejnovější. [11]

3 Analytické programování

Metoda *AP* vznikla v roce 2005. Na rozdíl od výše popsaných metod není tato metoda vázáná k jednomu evolučnímu algoritmu a taky není propojena s žádnou gramatikou či stromovou reprezentací. Lze ji použít s jakýmkoliv evolučním algoritmem. Je tomu tak, protože *AP* není samostatný evoluční algoritmus, ale rozhraní, které zobrazuje základní symbolické objekty do množiny možných řešení. Tato metoda je experimentální a slouží jako alternativní přístup ke genetickému programování nebo gramatické evoluci. [11]

3.1 Princip

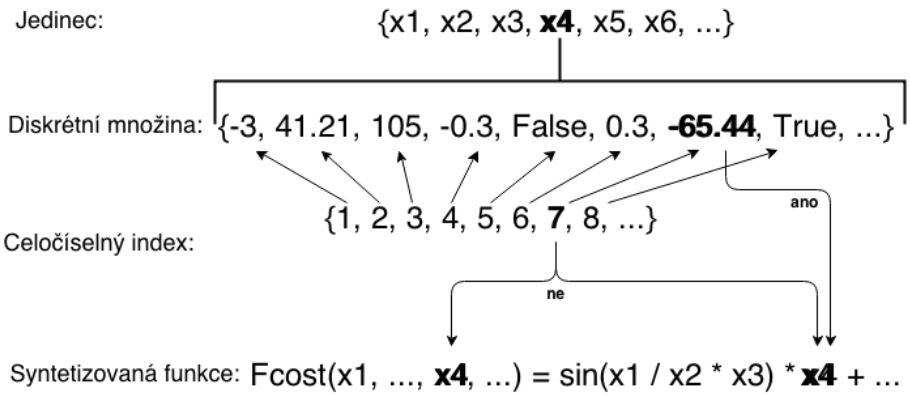
AP pracuje s množinou funkcí, operátoru a terminálních hodnot, která se nazývá General functional set (*GFS*). Tyto prvky množiny představují stavební kameny, ze kterých *AP* staví výsledné řešení. Prvky v této množině jsou hierarchicky uspořádány podle počtu svých argumentů do podmnožin. Každý z těchto prvků má svůj unikátní celočíselný diskrétní index, který reprezentuje daný objekt v *GFS*. V rámci množiny *GFS* můžou být definované jakékoliv uživatelské funkce, které uživatel potřebuje zohlednit při řešení problému. Na obrázku 6 je zobrazena množina *GFS*, včetně podmnožin, které mají označení GFS_{all}, GFS_{2arg}, GFS_{1arg} a GFS_{0arg} na základě počtu jejich argumentů. [11,12]



Obrázek 6: Množina GFS

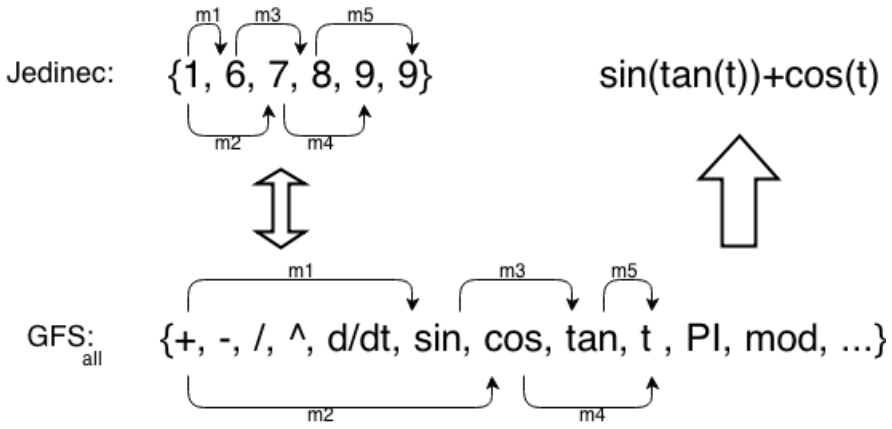
Jedinec v populaci je reprezentován na základě množiny indexů objektů z *GFS* množiny. Při jeho syntéze se za daný argument, tedy index, dosadí skutečný objekt z *GFS*, který tento index zastupuje. Tato technika se nazývá *DSH* a umožňuje numericky manipulovat s nenumerickými objekty. [4]

Tento princip je zobrazen na obrázku 7. Je na něm zobrazen jedinec při syntéze jeho čtvrtého parametru, který nabývá hodnoty 7. Tato hodnota je při sestavování funkce nahrazena odpovídající hodnotou z diskrétní množiny (*GFS*), tedy v tomto případě číslem na indexu 7, což je $-65,44$. Jedinci však můžou nabývat i reálných hodnot, ty ale musí být pro syntézu převedeny na indexy o rozsahu počtu prvků, které jsou v množině *GFS*.



Obrázek 7: DSH metoda

Syntéza celého jedince probíhá na základě obrázku 8. Je zde vidět podmnožina GFS_{all}, která obsahuje všechny objekty, ze kterých se jedinec může skládat. Dále je zde zobrazen konkrétní jedinec, kde jeho jednotlivé parametry jsou již převedeny na indexy, které reprezentují jednotlivé objekty z *GFS*. Syntéza začíná od prvního parametru jedince, což je v tomto případě hodnota 1. Protože se konec jedince nachází ještě daleko, je vybrán první objekt z množiny GFS_{all}. Objekt na indexu 1 v množině GFS_{all} je operátor $+$. Tento operátor požaduje dva argumenty, které je nutné vybrat. Protože má operátor dva argumenty, budou pro tuto funkci vybrány dva následující neobsazené prvky v jedinci. Tento výběr je v obrázku označen jako m_1 pro první parametr a m_2 pro druhý parametr. Jako první parametr operátoru $+$ bude vybrán objekt z *GFS*, který se nachází na pozici číslo 6, tedy funkce $sin()$. Druhý parametr operátoru $+$ bude představovat objekt na indexu 7, tedy funkce $cos()$. Zatím je tedy syntetizováno $sin() + cos()$, což bez parametrů funkcí $sin()$ a $cos()$ nelze vypočítat. Proto je potřeba v syntetizaci pokračovat a najít potřebné argumenty. Toto hledání je v obrázku 8 označeno jako m_3 a m_4. Po vyhledání těchto argumentů bude aktuální syntetizované řešení vypadat následovně: $sin(tan()) + cos(t)$. Aby tento výraz mohl být vypočítán, je potřeba definovat parametr funkce $tan()$. Z toho důvodu hledání parametrů dále pokračuje a v obrázku 8 je označeno jako m_5. Protože hodnota parametru, na který m_5 ukazuje, je rovna 9, tak je využit objekt z *GFS* na indexu 9, tedy terminál t. Protože již není co dál do výrazu doplnit, je výraz uzavřený a syntetizace končí. Výsledkem tedy je, že ekvivalentem k jedinci, reprezentovaného strukturou $\{1, 6, 7, 8, 9, 9\}$, je výraz $sin(tan(t)) + cos(t)$. [4] [12] [11]



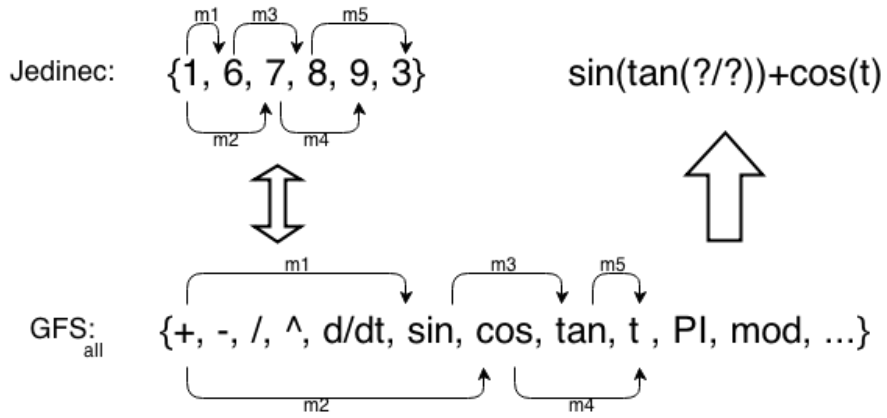
Obrázek 8: Syntetizace výsledku z jedince

To, jak bude výsledný výraz vypadat, závisí jednak na struktuře jedince, ale hlavně na pořadí objektů v množině *GFS*. Pokud se totiž změní pořadí objektů v této množině, tak se změní indexy jednotlivých objektů této množiny, takže stejný jedinec při použití různých *GFS* bude jinak syntetizován. [4]

3.2 Bezpečnostní procedury

Ne vždy však syntéza jedince proběhne v pořádku. Může se stát, že se při syntetizaci výrazů nepodaří nalézt všechny potřebné parametry pro syntetizované funkce a vzniknou tak nekompletní patologičtí jedinci, kteří reprezentují například neexistující funkce. Aby k tomuto jevu nedocházelo, obsahuje *AP* bezpečnostní procedury, které tyto problémy eliminují. [4]

Vychází se z toho, že jádrem *AP* je množina symbolických objektů *GFS*, která je rozdělena na podmnožiny podle počtu argumentů funkcí v nich obsažených. Při syntetizaci jedince se pak neustále měří vzdálenost do jeho konce. Na základě této změřené vzdálenosti se určuje, z jaké podmnožiny *GFS* se budou vybírat jednotlivé funkce pro syntetizaci. Pokud se při syntetizaci jedince blíží jeho konec, tak se další argumenty vybírají z podmnožiny *GFS*, která má menši počet argumentů než stávající podmnožina. Tím je zajištěno, že se při konci syntetizace budou jako argumenty funkcí vybírat terminály a tak nebudou vznikat patologické syntézy. Obrázek 9 demonstruje situaci, která by nastala, pokud by tato bezpečnostní procedura nebyla aplikována. Je zde vidět jedinec, u kterého při jeho syntéze dojde k tomu, že jsou využity všechny jeho parametry, ale výsledný výraz není úplný, takže nejde vypočítat a je tedy patologický. Otazníky v tomto výrazu symbolizují pozice chybějících argumentů. Pokud nastane situace, kdy má jedinec moc parametrů a všechny jeho parametry se pro syntézu nevyužijí, tak se nic neděje, protože pokud je jeho výraz uzavřen, jedná se o nepatologického jedince, u kterého však nebyly využity všechny jeho parametry. [12]



Obrázek 9: Problém při syntetizaci výsledku z jedince

3.3 Evoluční operace

Součástí *AP* jsou také samozřejmě evoluční operace jako křížení a mutace. Jejich použití je ale závislé na tom, jaký evoluční algoritmus *AP* používá. *AP* je totiž pouze rozhraní, které je schopno převést množinu vybraných objektů do výsledného řešení problému. Na úspěch *AP* má tedy největší vliv, jaký evoluční algoritmus je v kombinaci s *AP* použit. [11]

3.4 Posílené hledání

Aby *AP* vykazovalo ještě lepší výsledky, je možné použít techniku s názvem posílené hledání. Její princip spočívá v tom, že se do množiny objektů *GFS*, ze kterých jsou stavěny výsledná řešení, přidá aktuálně syntetizované řešení, které splňuje předem definovanou kvalitu. S tímto přidaným řešením se následně pracuje tak, jako s ostatními prvky *GFS*. Přidáním této nové informace do *GFS* se výkonnost *AP* zlepší. Pokud je však nalezeno nové řešení, které svou kvalitou překračuje řešení, které bylo do *GFS* již zavedeno, tak je toto staré řešení nahrazeno novým. Tím se výkonnost *AP* ještě více zkvalitní. Takto zavedené řešení se chápe jako terminál. [11] [12]

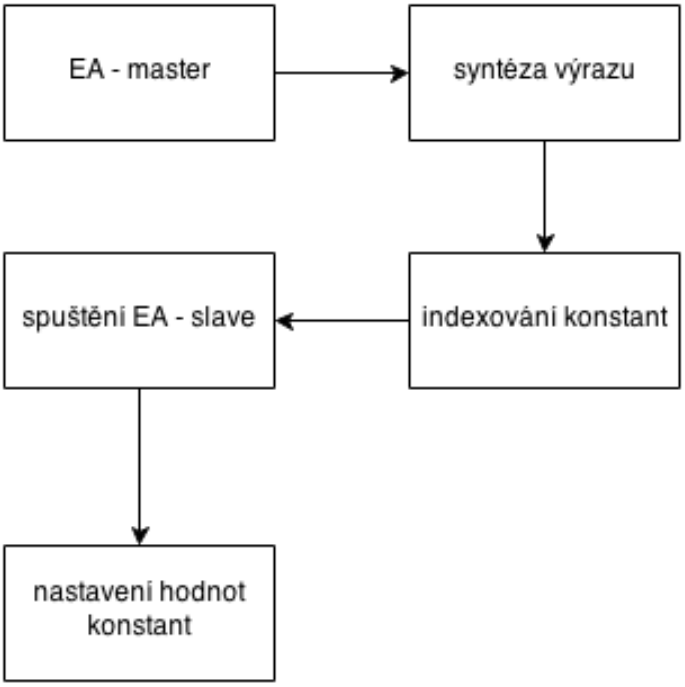
3.5 Meta evoluce

V situacích, kdy je potřeba počítat s tím, aby se v syntetizovaném výrazu vyskytovaly konstanty, nastává problém. Tyto konstanty totiž musí být podle principu *AP* součástí *GFS*, což pro konstantu, která má definiční obor definovaný v rámci všech reálných čísel představuje, že v *GFS* budou všechny hodnoty, kterých může tato konstanta nabývat. Tím velice naroste kardinalita této množiny. Pokud v ní je například 5 objektů, lze z ní vytvořit $5!$ možných výsledků. V případě, že její kardinalita naroste například na hodnotu 50, tak je to již $50!$ možných výsledků, což je obrovské číslo. Z tohoto důvodu je problém vyřešen jiným principem. Do *GFS* je přidán jeden terminální objekt, který představuje obecnou konstantu. Tím se kardinalita množiny zvýšila pouze o jedna. Tento přidaný objekt je

zapojen do syntézy výrazu. Po samotné syntéze se ve výrazu vyhledají všechny objekty, které představují tuto obecnou konstantu a oindexují se. Toto demonstruje výraz 3. [4]

$$tan(sin(x * K[1]) - K[2]) * sin(K[3]) \tag{3}$$

Hodnoty těchto konstant se následně určí pomocí podřízené evoluce, která se stará o jejich optimalizaci. Celý tento proces je zobrazen na obrázku 10, kde lze vidět jednotlivé kroky, které se při této činnosti provádějí. Tato verze AP s podřízenou evolucí pro odhad konstant se nazývá AP_{meta}. Při jejím použití je potřeba počítat s tím, že je časově náročná. [4]



Obrázek 10: Kroky při meta-evoluci

4 Použité evoluční algoritmy

V této práci jsem pro *AP* použil celkem základních 5 typů algoritmů spolu s možnými mutacemi těchto algoritmu. Tato kapitola se zaměřuje na představení těchto algoritmů a vysvětlení jejich principu, případně jejich možných variant.

4.1 Evoluční strategie (*ES*)

Tento algoritmus vznikl v 60. letech na Technické univerzitě v Berlíně jako experimentální optimalizační metoda. Poprvé byl aplikován v oblasti strojního inženýrství na problémy funkčního designu. Tento algoritmus existuje v několika variantách, které byly postupem času vyvinuty. Tou nejstarší jsou dvoučlenné *ES*, které byly vyvinuty a použity již v době, kdy nebylo dostatek výpočetního výkonu. Jejich časová náročnost tedy odpovídala výpočetnímu výkonu té doby. Po první aplikaci *ES* v počítači začaly vznikat další varianty tohoto algoritmu, jako jsou vícečlenné *ES* nebo rekombinační *ES*. Ve všech algoritmech *ES* je jedinec reprezentován v oboru reálných čísel kde pro jejich evoluci se vůbec nepoužívají operátory křížení ale pouze operátory selekce a mutace. [13]

4.1.1 Dvoučlenné ES

Tato varianta *ES* je nejjednodušší, protože pracuje pouze s jedním jedincem. Princip spočívá v tom, že se na základě směrodatné odchylky pro Gaussovo normální rozdělení vypočítá přírůstek, který je přičten k jedinci. Tím vznikne potomek, který se porovná pomocí cost value(*CV*) s jeho rodičem a pokud je kvalitnější, tak rodiče nahradí. Následně se celý princip iterativně opakuje. Přírůstek reprezentuje mutaci jedince, která je na základě Gaussova normálního rozdělení náhodná. Algoritmus je ukončen, pokud je nalezen jedinec odpovídající definované kvalitě nebo je dosažen maximální počet iterací. Tento algoritmus má parametery definováne v tabulce 2. [4, 14]

X	Počáteční náhodně vygenerovaný rodič
σ	Směrodatná odchylka pro Gaussovo normální rozdělení
f_{cost}	Účelová funkce vracející kvalitu řešení
$Iterace$	Maximální počet iterací algoritmu
FV	Kvalita jedince, při které se má algoritmus zastavit

Tabulka 2: Parametry ES

4.1.2 Vícečlenné ES

Postupem času se ukázalo, že je mnohem výkonnější, když se místo jednoho jedince (v případě dvoučlenného *ES*) použije jedinců více, tedy jejich populace. [4]

Pro mutaci jedinců se využívá stejný princip jako u dvoučlenných *ES*. Rozdíl je však v tom, že místo toho, aby v jednom cyklu algoritmu vznikl pouze jeden potomek, tak vznikne celá nová populace. Tato nová populace potomků se následně sjednotí s populací

rodičů. Tím vznikne jedna obrovská pomocná populace. Z ní se potom vyberou nejlepší řešení, které budou tvořit populaci rodičů v dalším cyklu algoritmu. Toto sjednocení populací do pomocné populace lze reprezentovat výrazem 4. [4]

$$P = \mu \cup \lambda = (\bigcup_{j=1}^{\lambda} y^j) \cup (\bigcup_{i=1}^{\mu} x^i) \tag{4}$$

Proměnná y^j reprezentuje j-tého jedince z populace potomků, která obsahuje λ jedinců. Konkrétní i-tý jedinec z populace rodičů o velikosti μ je reprezentován pomocí x^i. Jedná se tedy o takzvaný elitismus. Do nové rodičovské populace jsou totiž díky tomuto sjednocení vybíráni jak rodiče, tak potomci a to na základě jejich kvality. Tento algoritmus může existovat ještě v jedné variantě a to bez elitismu. Liší se ve způsobu naplnění nové populace, kdy pomocná populace je vytvořena podle vztahu 5. Proměnná λ zde představuje velikost populace potomků a proměnná y^j reprezentuje j-tého jedince této populace. [4, 15]

$$P = (\bigcup_{j=1}^{\lambda} y^j) \tag{5}$$

4.1.3 Rekombinační ES

Tato strategie přistupuje k mutaci jedince odlišným způsobem. Místo toho, aby byl potomek přímo vytvořen na základě rodiče a Gaussova normálního rozdělení, tak je nejprve vytvořen předpotomek, ze kterého je následně pomocí Gaussova normálního rozdělení vytvořen potomek, který již patří do nové populace. Tento předpotomek nese název rekombinant. Je vytvořen rekombinací několika rodičů, kde jejich počet je dán parametrem. Minimální hodnota tohoto parametru je 1. Maximální hodnota odpovídá velikosti populace. Pokud je tento parametr nastaven na hodnotu 1, tak se rekombinace ve skutečnosti neuskutečňuje, protože se operace účastní pouze jeden jedinec. Klíčové je, jakým způsobem je rekombinant z rodičů vytvořen. S touto strategií jsou spojovány dva typy rekombinací. První z nich je průměrová, kde je rekombinant vytvořen dle vztahu 6. Konstanta p zde reprezentuje počet jedinců pro rekombinaci. Předpotomek je tedy vypočten jako průměr z jeho rodičů. [4, 15]

$$y_{rek} = \frac{1}{p} \sum_{i=1}^{p} x^i \tag{6}$$

Druhým způsobem, jak vytvořit rekombinant, je pomocí diskrétní rekombinace. Předpotomek je pak vytvořen způsobem, že každý jeho parametr je náhodně vybrán z jeho rodičů. Pokud má tedy předpotomek dva parametry, tak první z nich bude mít stejnou hodnotu, jakou má jeden z jeho rekombinačních rodičů na prvním parametru. Druhý

parametr předpotomka bude zase tvořen druhým parametrem jednoho z jeho rekombinačních rodičů. Tuto situaci demonstruje tabulka 3. [4]

$Rodi_1$	$\mathbf{x_1^1}$	x_2^1
$Rodi_2$	x_1^2	x_2^2
$Rodi_3$	x_1^3	$\mathbf{x_2^3}$
$\mathbf{y_{rek}}$	$\mathbf{x_1^1}$	$\mathbf{x_2^3}$

Tabulka 3: Vytvoření rekombinantu pomocí diskrétní rekombinace

4.2 Diferenciální evoluce (*DE*)

Tento algoritmus byl poprvé vyvinut Kenem Pricem a Rainerem Stormem. Jedná se o poměrně nový algoritmus, který je známy od roku 1995. Je založen na tvorbě nových potomků na základě čtyř rodičů. Zvláštností je, že v rámci jeho cyklu probíhá mutace před křížením. [16, 17]

4.2.1 Parametry

Pro nastavení tohoto algoritmu je potřeba definovat řídící a ukončovací parametry, které mají vliv na činnost algoritmu a kvalitu výsledného řešení.

- **CR** - Tento řídící parametr představuje práh křížení. Jeho hodnota musí ležet v intervalu $[0, 1]$. Pokud je tato hodnota velice malá, tak se mutace skoro neprojeví ve zkušebním jedinci, který díky tomu bude pouze kopií aktuálního jedince a algoritmus bude stagnovat. V opačném případě, pokud bude hodnota *CR* nastavena na hodnotu blízké jedničce, tak bude zkušební jedinec vytvořen pouze ze tří náhodně vybraných rodičů. Algoritmus se díky tomuto nastavení bude chovat více stochasticky a evoluční chování bude potlačeno. [18]

- **D** - Tento řídící parametr je definován problémem, který je algoritmem řešen. Odvíjí se od něho tedy počet parametrů účelové funkce.

- **NP** - Definuje velikost populace, se kterou algoritmus pracuje. Hodnota tohoto parametru by neměla být menší než 4, protože tento počet je nejmenší možný, se kterým je algoritmus schopen pracovat. Obecně platí, že dobrý odhad tohoto parametru je mezi desetinásobkem a stonásobkem dimenze řešeného problému. [19]

- **F** - Tento řídící parametr představuje mutační konstantu. Její hodnota se pohybuje v intervalu $[0, 2]$. [18]

- **Generations** - Tento ukončovací parametr udává počet evolučních cyklů algoritmu. [4]

4.2.2 Populace

Algoritmus se v práci s populací oproti ostatním evolučním algoritmům moc neliší. Tvorba populace a ošetření parametrů je vyřešeno stejným způsobem, jak je popsáno v kapitole o evolučním cyklu (2.2).

4.2.3 Mutace

Algoritmus *DE* je z pohledu mutace zvláštní v tom, že pro vytvoření dalšího potomka je potřeba čtyř rodičů. Mutace probíhá tak, že se pro jedince, který má být zmutován, vyberou z populace tří nestejní jedinci. Pomocí těchto vybraných jedinců se vytvoří šumový vektor. Jeho tvorbu reprezentuje vztah 7. [16, 17]

$$y = x_3 + F(x_1 - x_2) \tag{7}$$

Rozdíl mezi prvním a druhým náhodně vybraným jedincem je vynásoben mutační konstantou *F*. Tento výsledný vektor je následně přičten k třetímu náhodně vybranému jedinci.

4.2.4 Křížení

Evoluční operátor křížení v *DE* nastává až po samotné mutaci, tedy po tvorbě šumového vektoru. Její princip spočívá ve využití tohoto šumového vektoru spolu s posledním nevyužitým rodičem. Vytvoří se z nich takzvaný zkušební vektor. Tento vektor je vytvořen s pomocí parametru prahu křížení *CR*. V cyklu se vybírají korespondující parametry z šumového vektoru a posledního nevyužitého rodiče. Pro každou dvojci parametrů je vygenerováno náhodné číslo. Pokud je jeho hodnota menší než hodnota *CR*, tak se do korespondujícího parametru zkušebního jedince přesune parametr z šumového vektoru. V případě, že by hodnota náhodného čísla byla větší nebo rovna hodnotě *CR*, tak se ve zkušebním jedinci uplatní parametr z posledního nevyužitého jedince. Tím je vytvořen nový jedinec, který o své místo v nastávající populaci soutěží s posledním nevyužitým rodičem, který byl použit pro tvorbu zkušebního vektoru. Výherce této soutěže představuje jednoho jedince v nové populaci. [16, 17]

4.2.5 Princip

Během činnosti algoritmu *DE* je prováděno šlechtění jedinců podle následujících kroků, které se provádějí během každé generace [18]:

1. **Stanovení parametrů** – Pro chod algoritmu je potřeba definovat všechny řídící a ukončovací parametry včetně vzorového jedince, který určuje, jak má jedinec z populace vypadat.
2. **Tvorba populace** – Je vytvořena vygenerováním náhodné populace na základě vzorového jedince.

3. **Započetí cyklu generace** – tento cyklus slouží pro postupné vybírání jedinců z aktuální populace, kde pro každého takto vybraného jedince se provede bod 4, tedy evoluční cyklus.

4. **Evoluční cyklus** – pro aktuálně vybraného jedince z populace se provede mutace a křížení, jak již bylo popsáno v 4.2.3 a 4.2.4. Jedinec, který na základě těchto evolučních operátorů vznikne, je zařazen do nově vznikající populace.

5. **Testování naplnění ukončovacích parametrů** – Algoritmus *DE* má pouze jeden ukončovací parametr a tím je počet generací. Algoritmus tedy bude ukončen, pokud je dosaženo tohoto počtu generací.

6. **Vyhodnocení** – Při jednotlivých generacích se v paměti neustále uchovává hodnota účelové funkce nejlepšího jedince. S touto hodnotou lze pracovat ve smyslu zachycení vývoje této hodnoty.

4.3 SOMA

Tento *EA* existuje od roku 1999. Jeho zkratka znamená „Samo Organizující se Migrační Algoritmus“. Je založen na vektorových operacích, které pracují s populací jedinců. Algoritmus se trochu vymyká standardnímu evolučnímu cyklu, protože se zde netvoří noví potomci a tak zde neexistují generace. Místo toho jedinci kooperativně migrují po prostoru možných řešení po takzvaných migračních kolech. Jeho myšlenka je totiž založena na napodobování chování skupiny inteligentních jedinců v přírodě. Tyto skupiny inteligentních jedinců spolu spolupracují za účelem vyřešení problémů. Může se jednat například o nalezení zdroje potravy nebo zdroje zlata. Algoritmus je tedy velice inspirovaný přírodou. Jeho vlastnost samo-organizace pochází z faktu, že se jedinci během hledání lepšího řešení navzájem ovlivňují. Tak v prostoru možných řešení vznikají skupiny jedinců, které připomínají putování živočichů v přírodě. [20]

4.3.1 Parametry

Před spuštěním algoritmu je nutné definovat jeho parametry. Ty lze rozdělit do dvou skupin. Jednak jsou to řídící parametry, které mají přímý vliv na migraci populace, ale také ukončovací, které definují, kdy bude činnost algoritmu ukončena. Skupiny těchto parametrů vycházejí z evolučního cyklu. Kvalita běhu celého algoritmu citelně závisí na jejich nastavení. [20]

- **PathLength** - Tento řídící parametr udává, jakou vzdálenost aktuální jedinec urazí směrem k jedinci, ke kterému směřuje. Pokud je hodnota tohoto parametru rovna jedné, tak aktuální jedinec doputuje přímo na místo cílového jedince. Je li tato hodnota rovná dvěma, pak doputuje na místo, které je za ním ve stejné vzdálenosti, ze které odstartoval, tedy dvojnásobek této cesty. V případě, že je hodnota parametru menší než jedna, tak k cílovému jedinci nedoputuje, ale zastaví se někde před ním. Tím však dojde ke znehodnocení migračního procesu. Doporučená hodnota tohoto parametru se pohybuje v intervalu $(1, 5]$. [4]

- **Step** - Jedná se o řídící parametr, který určuje zrnitost, s jakou bude zkoumána cesta aktuálního jedince směrem k cílovému. V případě, že je tato hodnota nastavena na malé číslo, tak bude cesta zkoumána velice podrobně. Naopak, pokud bude hodnota moc velká, tak se cesta prozkoumá v málo místech. Čím menší je hodnota parametru, tím je prohledávání prostoru výpočetně náročnější a zvyšuje se pravděpodobnost nalezení globálního extrému. Hodnota kroku by neměla být celočíselným násobkem vzdálenosti mezi aktuálním a cílovým jedincem. Doporučená hodnota parametru se pohybuje v intervalu $(0.11, PathLength]$. [4, 20]

- **PRT** - Jedná se o řídící parametr, podle kterého je vytvořen perturbační vektor. Zkratka *PRT* znamená perturbaci. Jeho hodnota ovlivňuje, kterým směrem se bude pohybovat aktuální jedinec vzhledem k cílovému jedinci. Hodnota tohoto parametru se pohybuje v intervalu $[0, 1]$. Pokud je vysoká, tak má *SOMA* vysokou tendenci konvergovat k lokálním extrémům. V případě, že je hodnota rovna jedné, tak stochastická složka algoritmu zaniká. Perturbační vektor je pro každého jedince vytvářen zvlášť a je platný pouze pro jeho jednu migraci. [20]

- **Dimenze** - Udává počet argumentů účelové funkce. Hodnota je tedy definovaná problémem, který má algoritmus řešit. Jedná se o řídící parametr. [4]

- **PopSize** - Jedná se o řídící parametr, který definuje, kolik jedinců bude populace obsahovat. Tato hodnota by měla být minimálně 10 a to z důvodu výkonnosti algoritmu. Maximální hodnota je definována uživatelem. [20]

- **Migrace** - Jedná se o ukončovací parametr, který říká, kolik bude provedeno migračních kol. V kontextu obecného evolučního cyklu se jedná o synonymum ke generacím. Počet migrací by se měl nacházet v intervalu $[10, ?]$, kde horní hranice intervalu je definována uživatelem. [20]

- **MinDiv** - Tento ukončovací parametr definuje na základě rozdílu mezi nejhorším a nejlepším jedincem v populaci, zda má být algoritmus ukončen. Ukončení nastane, pokud je tento rozdíl menší než hodnota parametru MinDiv. Pokud je parametr nastaven na velice malou hodnotu, tak se většinou algoritmus zastaví až po všech migracích. V případě, že je hodnota moc vysoká, je algoritmus často zastaven předčasně, než jedinci stačí nalézt globální extrém. [20]

4.3.2 Populace

Algoritmus se v práci s populací oproti ostatním *EA* moc neliší. Tvorba populace a ošetření parametrů je vyřešena stejným způsobem, jak je popsáno v kapitole o evolučním cyklu 2.2. [4]

4.3.3 Mutace

Mutace je důležitou součástí evolučního cyklu. Je založená na náhodě a jejím cílem je změnit některé vlastnosti jedince. V terminologii algoritmu *SOMA* se mutace nazývá per-

turbací a její míra závisí na nastavení *PRT* parametru. Na základě tohoto parametru se vytvoří *PRTVector*, který říká, který konkrétní parametr v jedinci bude vyrušen. Tento vektor se generuje pro každého jedince zvlášť a je platný pouze pro jednu migraci aktivního jedince. Perturbační vektor je vytvořen na základě kódu 1. Pro každý jeho parametr se vygeneruje pomocí generátoru náhodných čísel náhodné číslo v intervalu $[0, 1]$, které se porovná s *PRT* parametrem. Pokud je toto náhodné číslo menší než hodnota *PRT* parametru, tak je k danému prvku perturbačního vektoru přiřazena hodnota 1. V opačném případě 0. [20]

```
for(j =1; i<N; j++) {
    if (randomj < PRT) {
        PRTVectorj = 1
    } else {
        PRTVectorj = 0
    }
}
```

Výpis 1: Kód pro vytvoření PRTVectoru

Tabulka 4 zobrazuje vytvoření *PRTVectoru* pro jedince o třech parametrech, kde hodnota parametru *PRT* je nastavena na 0.2.

Nhodnslo	**PRTVector**
$0,152$	**1**
$0,283$	**0**
$0,451$	**0**

Tabulka 4: Vytvoření perturbačního vektoru

4.3.4 Křížení

Klasický přístup k evolučnímu operátoru křížení představuje vygenerování nového potomka na základě již existujících rodičů. Tento potomek může být kvalitnější nebo méně kvalitní než jeho rodiče a na základě této hodnoty, buď je, nebo není do nové populace zahrnut. Algoritmus *SOMA* však provádí křížení odlišným způsobem. Při putování aktuálního jedince k cílovému, je prostor možných řešení prozkoumáván po diskrétních skocích. Tyto skoky reprezentují sekvenci potomků, z nichž jako konečný, je vybrán ten nejkvalitnější potomek. Každý jedinec si tedy pamatuje pozici, která během jeho cesty k cílovému jedinci měla největší kvalitu *CV*. Toto nejlepší řešení nahradí aktuálního jedince v dalším migračním kole. Cesta jedince je řízena vztahem 9. [20]

$$\overrightarrow{r} = \overrightarrow{r_0} + \overrightarrow{m} * t * PRTVector \tag{8}$$

$$t \in < 0, po\ Step\ až\ do, PathLength > \tag{9}$$

Vektor $\overrightarrow{m}$ ve vztahu 9 představuje směrový vektor, který směřuje od místa aktuálního jedince směrem k cílovému jedinci. Parametr t může nabývat pouze tolika hodnot, kolikrát

se parametr *Step* vejde do parametru *PathLength*. Vektor r_0 určuje pozici aktuálního jedince. Z výrazu tedy lze vidět, že *PRTVektor* ovlivňuje pouze druhý sčítanec výrazu, tedy směr. Pokud jsou všechny prvky tohoto vektoru rovny 1, pak vektor r míří přímo směrem na cílového jedince. V případě, že některý prvek *PRTVektoru* je nulový, tak jsou díky vztahu násobení vynulovány i odpovídající prvky vektoru m. Tím pádem se hodnoty odpovídajících souřadnic nemění.

4.3.5 Princip

Jak již bylo řečeno, algoritmus *SOMA* je inspirován chováním inteligentních jedinců v přírodě, kteří spolu spolupracují, aby vyřešili společný problém. Tito inteligentní jedinci můžou být například smečka lovících vlků nebo kolonie termitů. U těchto příkladů je společným jmenovatelem nalezení potravy. Tito jedinci spolu spolupracují, ale i zároveň soutěží. Každý chce být totiž ten, který nalezne nejlepší kořist. Aby se zjistilo, kdo tuto nejlepší kořist nalezl, musí si jedinci navzájem sdělit, jak kvalitní kořist našli. Tomuto předávání informací se říká fáze spolupráce. Na základě těchto informací se snaží přizpůsobit své chování tak, aby při dalším předávaní informací s ostatními nalezli co nejlepší kořist, tedy vyhráli soutěž. Výherce soutěže se totiž na rozdíl od svých kolegů nemusí kořisti vzdát. Všichni ostatní jedinci se své kořisti musí vzdát a migrovat, čímž začíná opět soutěžení o nejlepší kořist. Tato činnost se opakuje, dokud všichni jedinci nenajdou nejlepší kořist. Takto ve zjednodušené podobě pracuje algoritmus *SOMA*. Při jednotlivých migračních kolech se tedy netvoří noví potomci, ale mění se pouze jejich pozice na základě výrazu 9. Vzorce chování, kterými se snaží jedinci doputovat k nejlepšímu řešení, se nazývají strategie. Základní verze algoritmu *SOMA* se strategií všichni k jednomu, se skládá z následujících kroků [4, 20]:

- **Definice parametrů** - Před spuštěním algoritmu je potřeba definovat již dříve zmíněné ukončovací a řídící parametry. Důležité je také vytvořit vzorového jedince, který představuje, jak má vypadat jedinec v populaci. Nutné je také předložit řešený problém ve formě účelové funkce, která slouží pro jedince jako „životní prostředí".
- **Tvorba populace** - V rámci tohoto kroku je na základě vzorového jedince vytvořena počáteční populace. Každý parametr v jedinci je tedy inicializován pomocí generátoru náhodných čísel.
- **Migrační kola** - Migrace jedinců začíná ohodnocením všech jedinců v populaci pomocí účelové funkce. Nejkvalitnější jedinec je zvolen jako leader pro následující migrační kolo. Nyní se všichni jedinci, kromě leadera, začnou pohybovat pomocí skoků směrem k leadrovi. Velikost skoku je závislá na řídícím parametru *Step*. Každý jedinec si po každém skoku přepočítá svojí kvalitu na základě účelové funkce. Pokud je lepší, než jeho kvalita, kterou měl předtím, tak si tuto novou hodnotu zapamatuje. Skoky směrem k leadrovi se zastaví, až je dosaženo pozice, která je dána parametrem *PathLength*. Po posledním skoku jedince se jedinec vrací na místo, kde byla nalezena během jeho cesty nejlepší kvalita na základě účelové funkce. Každý jedinec kromě leadera má tak po migračním kole novou pozici z prostoru možných řešení. Před

samotnou cestou k jedinci je vygenerován perturbační vektor o stejné dimenzi, jako je řešený problém. Tento vektor je na základě parametru *PRT* naplněn tak, že každý prvek z tohoto vektoru nabývá hodnoty 0 nebo 1. Ten je pak použit pro výpočet skoku jedince.

- **Testování naplnění ukončovacích parametrů** - Tento krok má na starost kontrolu naplnění ukončovacích parametrů. Testuje tedy, zda rozdíl mezi nejlepším a nejhorším jedincem je menší než ukončovací parametr *MinDiv* a zároveň, jestli proběhlo poslední migrační kolo. Pokud ani jedna z podmínek není splněna, tak algoritmus pokračuje krokem „Migrační kola".

- **Stop** - Tento krok má za úkol už pouze vrátit nejkvalitnějšího jedince z posledního migračního kola, který představuje nejlepší nalezené řešení řešeného problému.

4.3.6 Strategie

V algoritmu *SOMA* lze definovat několik strategií, podle kterých jedinec provádí migraci za lepším řešením. Tyto strategie se podílí na výběru skoků jedince v rámci migračního kola.

- **Všichni k jednomu** - Podle názvu lze usoudit, jak tato strategie funguje. Všichni jedinci kromě leadera migrují k leadrovi. Jedná se tedy o strategii, která byla popsána v předchozím textu v rámci principu algoritmu *SOMA* 4.3.5. [4]

- **Všichni ke všem** - Zvláštností této strategie je fakt, že zde neexistuje leader, ke kterému by ostatní jedinci migrovali. Místo toho migrují ke všem. Na svou novou nejlepší nalezenou pozici během migračního kola se ale přesunou až na začátku dalšího migračního kola před samotným začátkem migrace. Díky tomuto chování je zajištěno, že ostatní jedinci v populaci migrují k jedincům na pozicích, které nabývali před jejich samotnou migrací. Tato strategie je výpočetně náročnější, protože jedinec tímto způsobem musí prohledat mnohem více potencionálních nových pozic. Je však větší pravděpodobnost, že bude nalezen globální extrém, protože prostor možných řešení je prohledán více podrobněji, než v případě strategie „všichni k jednomu" o stejných parametrech. [4]

- **Adaptivně všichni ke všem** - Tato strategie je velice podobná strategii „Všichni ke všem". Rozdíl je však v tom, jakým způsobem jedinec migruje k ostatním jedincům. Aktuální jedinec totiž změní svojí pozici při každé migraci k jedinci, ke kterému ještě nemigroval. Pokud jsou v populaci tři jedinci, tak první jedinec bude v rámci jeho migračního kola migrovat nejprve k druhému jedinci z populace. Na této cestě se ihned přesune na nejlepší místo, které objevil při migraci k tomuto druhému jedinci. Ihned na to bude z tohoto nového místa migrovat k třetímu jedinci, kde se provede stejný postup. [4]

- **Všichni k jednomu náhodně** - Tato strategie je velice podobná strategii „Všichni k jednomu". Rozdíl je však ve způsobu výběru leadera, ke kterému budou jedinci

migrovat. Ve verzi „Všichni k jednomu“ byl dán nejkvalitnější hodnotou účelové funkce. V této strategii je pro každého jedince v každém migračním kole určen zcela náhodně. [4]

- **Svazky** - Jedná se o zvláštní strategii, která se dá aplikovat na jakoukoli z výše uvedených strategií. Její princip spočívá v rozdělení populace na pod-populace, kde v každé z nich probíhá samostatně algoritmus *SOMA*. Tyto svazky můžou být díky pohybu jedinců v prostoru možných řešení slučovány nebo se naopak můžou rozpadat. [4]

4.4 Rojení částic (*PS*)

Historie tohoto algoritmu začíná rokem 1995, kdy byl poprvé popsán Russellem Ederhartem a Jamesem Kennedym. Je inspirován chováním živočišných hejn, jako jsou například rybí hejna nebo ptačí hejna. Je tedy také jako například *SOMA* založen na populaci jedinců, ze které se snaží vytvářet pořád kvalitnější populace. Tento algoritmus ale neobsahuje evoluční operátory pro křížení a mutaci. Místo toho se noví jedinci tvoří tím, že v prostoru možných řešení následují trajektorie kvalitnějších částic. Právě díky této vlastnosti se migrace těchto jedinců jeví jako hejno. [4, 21]

4.4.1 Princip

Každého jedince z populace si lze představit jako ptáka, který se nachází v určité nadmořské výšce. Ten jedinec, který je v nejvyšší nadmořské výšce, je nejkvalitnější. Při každé iteraci algoritmu všichni z populace ví, který jedinec je nejkvalitnější a kde se v prostoru možných řešení nachází. Na základě této informace, všichni ptáci ví, kterým směrem se vydat, aby se k tomuto nejkvalitnějšímu jedinci dostali. V terminologii tohoto algoritmu se jedinec nazývá částice. Všechny tyto částice mají svojí pozici v prostoru možných řešení a rychlost, která definuje vlastnosti jejich pohybu v prostoru. Mimo tyto hodnoty si také pamatují svojí dosud nejlepší dosaženou pozici. Populace částic je vytvořena náhodně. Tato operace zahrnuje také náhodné vygenerování vektoru rychlosti, který udává jakým směrem se má částice v dalším kroku vydat. Částice je na základě její aktuální pozice ohodnocena hodnotou účelové funkce. Nejkvalitnější částice z této populace je následně uložena do společné paměti populace. Každý z jedinců tedy ví, kde se nachází nejkvalitnější částice populace. Tento jedinec se ve společné paměti populace označuje jako $gBest$. Každý z jedinců populace si následně musí ověřit, jestli jeho aktuální pozice není lepší než jeho dosud nejlepší dosažená pozice. Pokud tato situace nastane, tak si tuto aktuální pozici uloží do své paměti, která se označuje jako $pBest$. Na základě hodnot $gBest$ a $pBest$ si částice upraví svou rychlost a pozici na základě vztahů 10 a 11. [4, 22]

$$v_d(t+1) = v_c(t) + c_1 * rand * (pBest_{i,d} - x_{i,d}(t)) + c2 * rand * (gBest_d - x_{i,d}(t)) \quad (10)$$

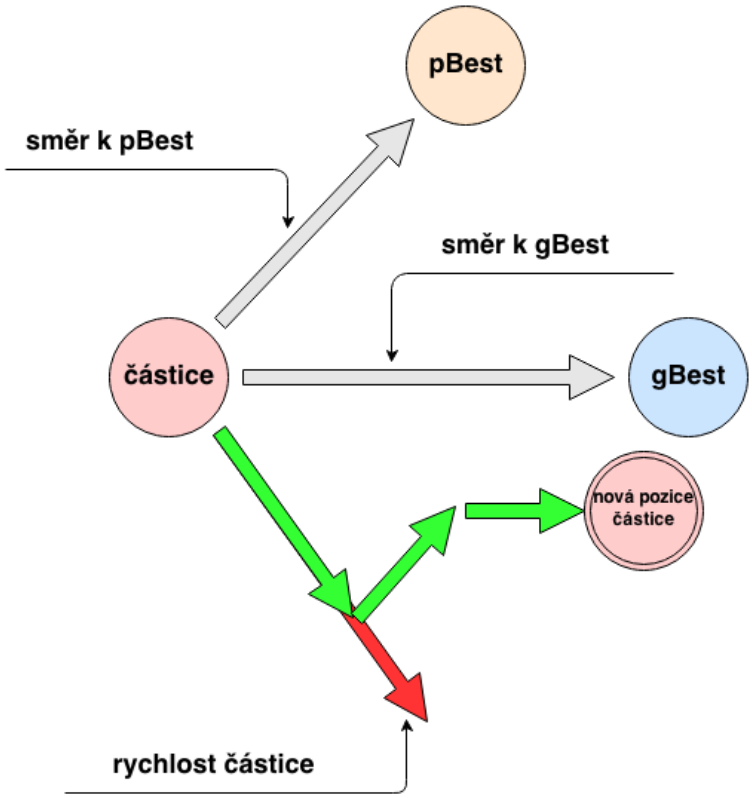
$$x_{i,d}(t+1) = x_{i,d}(t) + v_d(t+1) \quad (11)$$

Jednotlivé proměnné ve výrazech 10 a 11 mají význam dle tabulky 5:

$V_d(t+1)$	Rychlost jedince v následujícím kroku
$V_d(t)$	Rychlost jedince v tomto kroku
$x_{i,d}(t+1)$	Pozice jedince v následujícím kroku
$x_{i,d}(t)$	Pozice jedince v tomto kroku
$rand$	Náhodné číslo z intervalu $(0, 1)$
c_1, c_2	Učící faktory
$pBest_{i,d}$	Nejlepší dosažená pozice daného jedince
$gBest_d$	Nejlepší nalezená pozice v populaci

Tabulka 5: Význam proměnných u výpočtu rychlostního vektoru

Vztah 10 se skládá ze tří sčítanců, které ovlivňují výsledný směr rychlostního vektoru. První sčítanec představuje vektor s jeho aktuální pozicí. Druhý sčítanec představuje vektor, který se snaží mířit směrem k dosud nejlepší dosavadní pozici jedince. Třetí sčítanec představuje vektor, který se snaží rychlostní vektor jedince natočit více směrem ke globálnímu nejlepšímu řešení. Tento princip je zobrazen na obrázku 11.



Obrázek 11: Pohyb částice

Na základě takto vypočítaného rychlostního vektoru se jedinci přesunou na nově vypočítanou pozici. Následně se opět ověří jejich kvalita a celý cyklus výpočtu se opakuje.

Maximální délka rychlostního vektoru je omezena parametrem V_{max}. V případě, že délka rychlostního vektoru je větší, než tato definovaná mez, existují dvě řešení, jak tuto situaci řešit. První z nich je, že pro jedince je vygenerován nový rychlostní vektor. Druhá možnost je omezit hodnotu na V_{max}. Toto omezení zabraňuje tomu, aby částice rychle opustila prostor možných řešení. Pokud se tomu tak stane, je této částici náhodně vygenerována její nová pozice. [21]

4.4.2 Parametry

- **Dimenze** - Tento parametr je dán problémem, který má algoritmus optimalizovat. Je to tedy počet argumentů účelové funkce. [4]
- **Rozsah** - Udává velikost prohledávaného prostoru. Na základě vzorového jedince definuje jednotlivé rozsahy dimenzí. Lze tedy přímo specifikovat, kde má být řešení v prostoru hledáno. [4]
- **Počet částic** - Definuje velikost populace, tedy počet částic. Čím vyšší toto číslo bude, tím větší bude výpočetní náročnost. Prostor však bude prohledán více podrobně. Obvykle se toto číslo volí v rozsahu 20 až 40 částic. [4]
- $\mathbf{V}_{max}$ - Slouží pro omezení maximální rychlosti částice. Tato hodnota má vysoký vliv na důkladnost prohledávání. Pokud je maximální rychlost částice moc vysoká, tak se neprohledává důkladně, protože se částice rychle vzdaluje a často překročí povolené meze prohledávaného prostoru. To následně způsobí, že pro její pozici je vygenerována nová náhodná pozice a tím se stane tento algoritmus pouze náhodným prohledáváním, ve kterém je evoluce potlačena. V případě, že je maximální rychlost částice malá, je prohledávána pouze malá část prostoru, ale důkladně. Jako ideální hodnota maximální rychlosti se doporučuje používat jednu dvacetinu povoleného rozsahu. [21]
- **Učící faktory** $\mathbf{c}_1$**,** $\mathbf{c}_2$ - Tyto hodnoty ovlivňují prioritu pohybu částice. Faktor c_1 se podílí na míře pohybu částice směrem k jejímu dosud nejlepšímu řešení. Naopak faktor c_2 se podílí na míře pohybu částice směrem ke globálně nejlepšímu řešení. Jedná se tedy o váhy, které dávají jednotlivým směrům významnost. Oba tyto faktory jsou většinou nastaveny na hodnotu 2. [21]
- **Setrvačnost** - Parametr tohoto typu slouží pro specifikaci, jaké extrémy se mají hledat. Pokud je jeho hodnota malá, tak populace má tendenci konvergovat k lokálním extrémům. V opačném případě, pokud je velká, má populace tendenci konvergovat ke globálním extrémům. V rovnici je tímto parametrem násobena předcházející rychlost jedince. Ze strategického pohledu prohledávání prostoru však může být výhodné, aby se hodnota setrvačnosti postupně snižovala. Díky této vlastnosti algoritmus na začátku svého běhu prohledává prostor po velkých skocích, které postupně snižuje. Nízká hodnota setrvačnosti v pokročilém stádiu algoritmu dovoluje lépe prohledat bližší okolí nejlepšího jedince. Výpočet setrvačnosti na základě této

strategie probíhá podle vztahu 12. Proměnné w_{start} a w_{end} ze vztahu 12 představují počáteční a koncovou hodnotu setrvačnosti. [21]

$$w = w_{start} - \frac{(w_{start} - w_{end}) * iterace}{Migrace} \tag{12}$$

4.5 Simulované žíhání (*SA*)

Tento algoritmus byl nezávisle popsán Scottem Krikpatrickem v roce 1983 a Vladem Černým v roce 1985. Inspirace za tímto algoritmem pochází z metalurgie, kde žíhaný kov se zahřál na určitou teplotu a postupně se ochlazoval, čímž se dostával do rovnovážného stavu. Tím v kovu vznikaly velké krystaly s menšími defekty. Při zahřátí se totiž uvolnily atomy ze svých výchozích pozic v krystalové mřížce a začaly náhodně kmitat s vysokou energií. Během ochlazování se jejich energie snižuje a tak kmitají čím dál méně, až se ustálí na své nové pozici v krystalové mřížce a tak přejdou do rovnovážného stavu. Pokud je proces ochlazování velice pomalý, pak je těleso při každé teplotě schopno dosáhnout rovnovážného stavu. Pravděpodobnost, že se do tohoto stavu dostane, je dána Boltzmannovým rozdělením pravděpodobnosti podle výrazu 13, kde proměnná T z výrazu představuje aktuální teplotu a funkce $f(x)$ vrátí na základě stavu atomu jeho energi. [4, 23, 24]

$$\frac{e^{\frac{-f(x)}{T}}}{Q(T)} \tag{13}$$

$$Q(T) = \sum_{i} e^{\frac{-f(x)}{T}} \tag{14}$$

4.5.1 Princip

Algoritmus simulovaného žíhání si toto postupné ochlazování adaptoval. Žíhání začíná na počáteční teplotě, která se během jeho činnosti zmenšuje až na teplotu konečnou. Při každé teplotě probíhá Metropolisův algoritmus. Tento algoritmus je odpovědí jak aproximovat Boltzmannové rozdělení v počítači. Pří práci s vektory reálných čísel je totiž na základě výrazu 14 potřeba projít všechny možné stavy těchto vektorů, což není výpočetně možné. Zjistilo se, že celé Boltzmannové rozdělení, tedy včetně tohoto neimplementovatelného výpočtu, lze nahradit právě Metropolisovým algoritmem, který je proveden několikrát. Je založen na technice Monte Carlo a rozhoduje o tom, zda změnu stavu částice akceptovat nebo ne a to na základě vztahu 15. [23, 24]

$$f(x \to x_0) = \begin{cases} 1, & \text{pro } f(x) < f(x_0) \\ e^{-(f(x)-f(x_0))/T}, & \text{pro } f(x) \leq f(x_0) \end{cases} \tag{15}$$

Pokud má nový stav x menší funkční hodnotu než původní stav x_0, tak je nový stav akceptován. V opačném případě je akceptován pouze s určitou pravděpodobností danou vztahem. Nový vztah je pro každý běh algoritmu vybrán na základě normálního

rozdělení. Po ukončení běhů Metropolisova algoritmu pro aktuální teplotu, je aktuální teplota pomoci funkce $alpha$ snížena. Následně se celý cyklus opakuje, dokud není dosaženo konečné teploty. [24]

4.5.2 Parametry

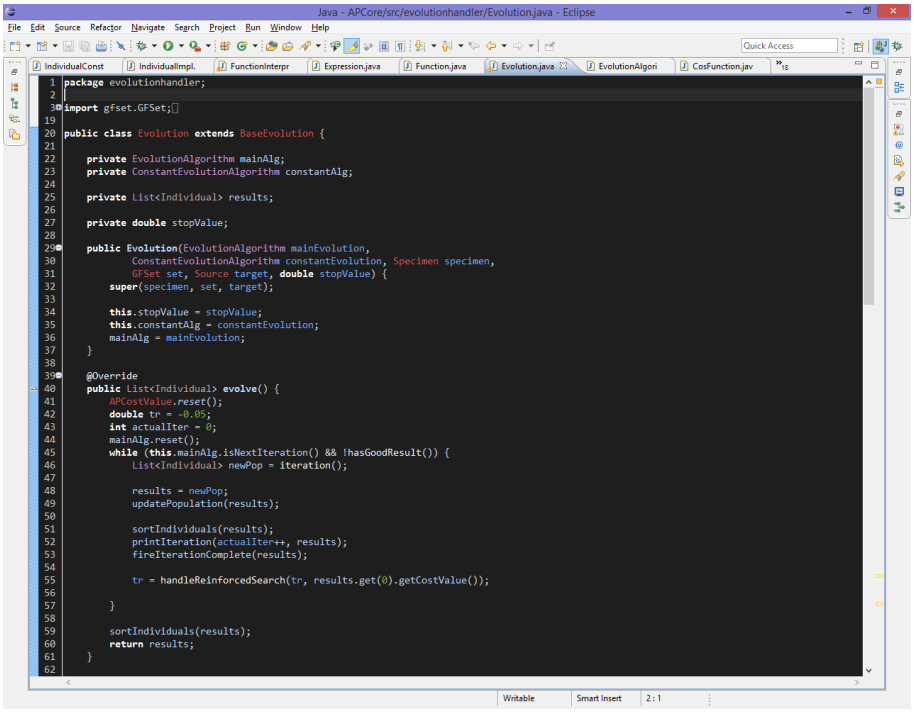
Pro správnou inicializaci algoritmu je potřeba definovat následující parametry:

- $\mathbf{T}_0$ - Počáteční teplota.
- $\mathbf{T}_f$ - Konečná teplota.
- $\mathbf{n}_T$ - Počet opakování Metropolisova algoritmu pro teplotu.
- **alpha** - Funkce pro redukci teploty, která každé teplotě přiřazuje teplotu o něco nižší.

5 Implementace

Pro implementaci programu je použit jazyk Java spolu s *IDE* Eclipse (obr. 12). Důvodem volby tohoto jazyka byla především jeho nezávislost na platformě, což vede k tomu, že výsledný program lze využít jak v systému Windows tak v OSX. Jedinou podmínkou je však mít nainstalován *JVM*. Při implementaci byl brán velký ohled na obecnost a možnou budoucí rozšiřitelnost programu. Uživatel si při práci s programem může sám nadefinovat, jaké algoritmy chce pro provedení *AP* použít, včetně široké konfigurace dostupné v konfiguračním souboru aplikace. [25]

Jako řešený problém jsem vybral prokládání bodů funkcí. Na základě definovaných bodů se program snaží vytvořit předpis funkce tak, aby co nejlépe kopíroval definované body. Aby bylo tyto body možné definovat efektivněji, je program schopen na základě předložené funkce pomocí vzorkování tyto body vyrobit. Tato kapitola slouží zároveň jako dokumentace programu.



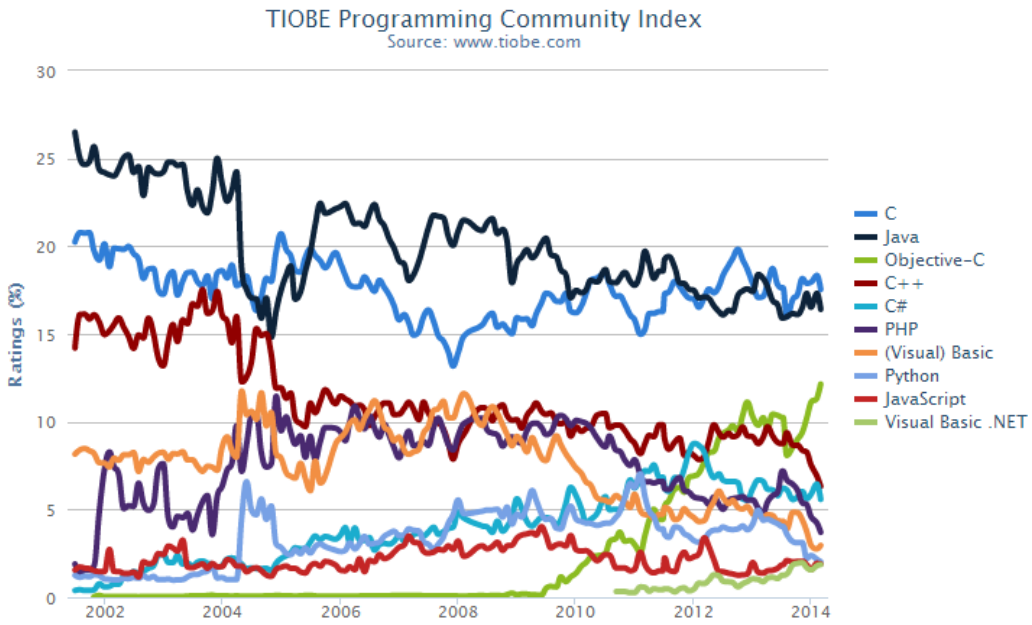

```
package evolutionhandler;

import gfset.GFSet;

public class Evolution extends BaseEvolution {

    private EvolutionAlgorithm mainAlg;
    private ConstantEvolutionAlgorithm constantAlg;

    private List<Individual> results;

    private double stopValue;

    public Evolution(EvolutionAlgorithm mainEvolution,
            ConstantEvolutionAlgorithm constantEvolution, Specimen specimen,
            GFSet set, Source target, double stopValue) {
        super(specimen, set, target);

        this.stopValue = stopValue;
        this.constantAlg = constantEvolution;
        mainAlg = mainEvolution;
    }

    @Override
    public List<Individual> evolve() {
        APCostValue.reset();
        double tr = -0.05;
        int actualIter = 0;
        mainAlg.reset();
        while (this.mainAlg.isNextIteration() && !hasGoodResult()) {
            List<Individual> newPop = iteration();

            results = newPop;
            updatePopulation(results);

            sortIndividuals(results);
            printIteration(actualIter++, results);
            fireIterationComplete(results);

            tr = handleReinforcedSearch(tr, results.get(0).getCostValue());

        }

        sortIndividuals(results);
        return results;
    }
```

Obrázek 12: IDE Eclipse

5.1 Jazyk Java

Java je objektově orientovaný jazyk od společnosti *Sun Microsystems*. Poprvé byl touto firmou představen 23. května roku 1995. Aktuálně je na základě *Tiobe* indexu (obr. 13) hodnocen jako jeden z nejpoužívanějších programovacích jazyků této doby. Jeho popularita však s nástupem nových programovacích jazyků mírně klesá. Obrovskou výhodou

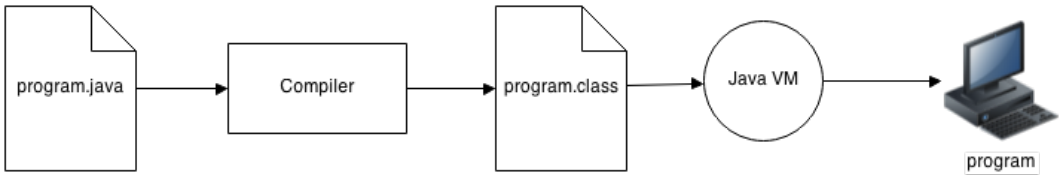
tohoto jazyka je jeho přenositelnost. Programy tak stačí napsat pouze jednou a lze je spustit na různých systémech přes mobilní zařízení, desktopové počítače až po distribuované systémy, které běží po celém světě. [25, 26]



Obrázek 13: Tiobe index. Obrázek převzat z [2]

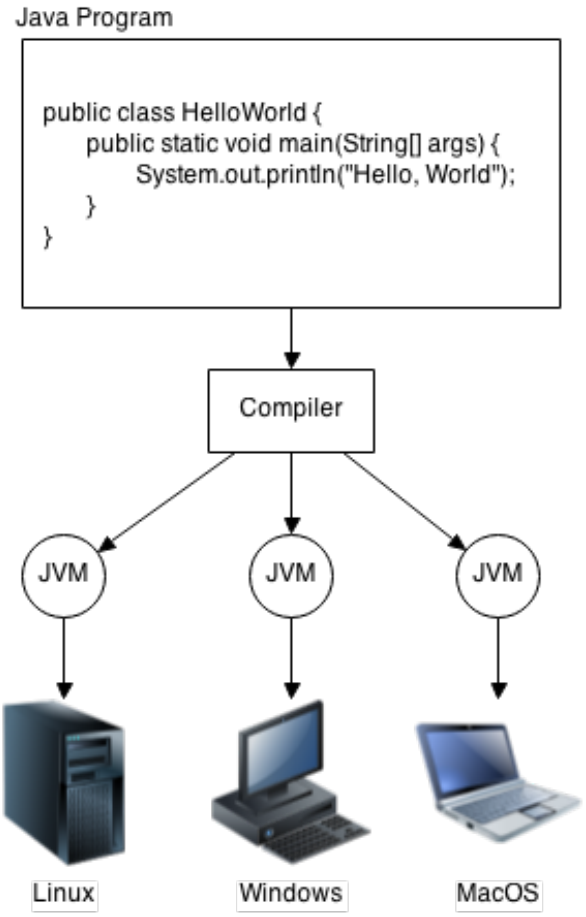
5.1.1 Architektura

Při programování je zdrojový kód aplikace nejprve napsán v čistém textovém souboru pomocí jazyka Java. Tyto soubory mají příponu *.java* a jsou následně zkompilovány pomoci *javac* kompilátoru do souborů s příponou *.class*. Tyto soubory obsahují takzvaný *bytecode*, což není kód určený speciálně pro aktuálně používaný procesor ale pro *JVM*. Java následně tyto soubory spouští jako instanci *JVM*. Tento proces reprezentuje obrázek 14. [27]



Obrázek 14: Kompilace zdrojových kódů v jazyce Java

Díky tomu, že *JVM* je dostupný pro mnoho různých platforem, mohou být soubory *.class* spuštěny na různých platformách jako je *Windows* nebo *Linux*. Tuto vlastnost demonstruje obrázek (15). [27]



Obrázek 15: Přenositelnost jako vlastnost Javy

Java tedy není pouze programovací jazyk, ale celá platforma. V tomto případě je to však pouze softwarová vrstva, která leží nad hardwarově orientovanou platformou. Platforma Java se skládá celkem ze dvou komponent: [27]

- Java Virtual Machine (*JVM*)
- Java Application Programming Interface (*Java API*)

JVM je základem platformy a je portován na mnoho *OS*. *Java API* je obrovská kolekce předem připravených kódů, které může programátor pro svou práci využívat. Je rozdělena do různých knihoven, které seskupují příbuzné kódy. Tyto knihovny se nazývají balíčky. Díky tomu, že se kód programu Java vykonává v *JVM* a ne přímo nativně, je běh kódu o trochu pomalejší, než je tomu třeba u jazyku *C*. [27]

5.2 Struktura projektu

Program je rozdělen do dvou komponent. První komponenta představuje jádro *AP*, které má za úkol provádět evoluci pomocí evolučních algoritmů. Druhá komponenta představuje klientské rozhraní, které toto jádro využívá a staví nad ním další vrstvu abstrakce tak, aby se toto jádro dalo uživatelem efektivně využít. Úkolem tohoto rozhraní je pak správně načíst konfiguraci ze souboru a správně nastavit jádro *AP*, případně vizuálně zobrazit výsledek. Následující tabulky zobrazují organizaci balíčků tohoto programu včetně vysvětlení, co balíčky obsahují. Balíčky v komponentě pro jádro AP jsou zobrazeny v tabulce 6. Balíčky v komponentě klientského rozhraní představuje tabulka 7.

Balíček	**Význam**
comparator	Obsahuje třídy pro porovnávání jedinců
costvalue	Obsahuje rozhraní pro práci s CV včetně implementací těchto rozhraní
evolutionalgorithm	Obsahuje rozhraní pro práci s EA
evolutionalgorithm.$*$	Zde se nacházejí implementace jednotlivých EA
evolutionhandler	Zde se nacházejí třídy definující obecný evoluční cyklus
exceptions	Balíček určen pro uživatelské vyjímky
generator	Obsahuje rozhraní pro generátor náhodných čísel včetně jeho implementací
gfset	Definuje GFS a základní rozhraní pro funkce v ní
gfset.functions.$*$	Zde se nachází implementace jednotlivých funkcí, které můžou být součásti GFS
helpers	Zde se nachází pomocné třídy pro běžnou práci
individual	Obsahuje všechno, co se týká jedinců včetně rozhraní, implementací tohoto rozhraní a generátoru jedinců
interpret	Zde se nachází třídy zodpovědné za převod jedince z množiny čísel do množiny řešení včetně potřebných rozhraní
sourcevalue	Tento balíček obsahuje třídy potřebné pro reprezentaci vstupní funkce
specimen	Balíček obsahuje vše, co se týká vytváření a definice vzorového jedince pro tvorbu populace

Tabulka 6: Balíčky v komponentě pro jádro AP

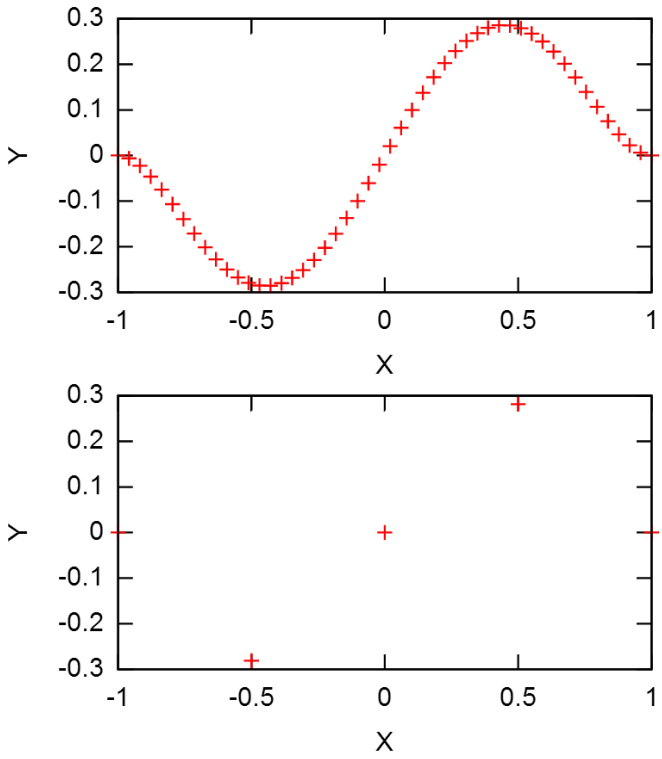
Balíček	Význam
tyl0010.client	Obsahuje třídy reprezentující klientský přístup k programu
tyl0010.client.config	Obsahuje vše pro interpretaci vstupního konfiguračního souboru
tyl0010.client.visualize	Obsahuje třídy pro vizualizaci
net.sourceforge.jeval	Balíček pro výpočet funkce zadané jako řetězec

Tabulka 7: Balíčky v komponentě pro klientské rozhraní

5.3 Reprezentace vstupní funkce

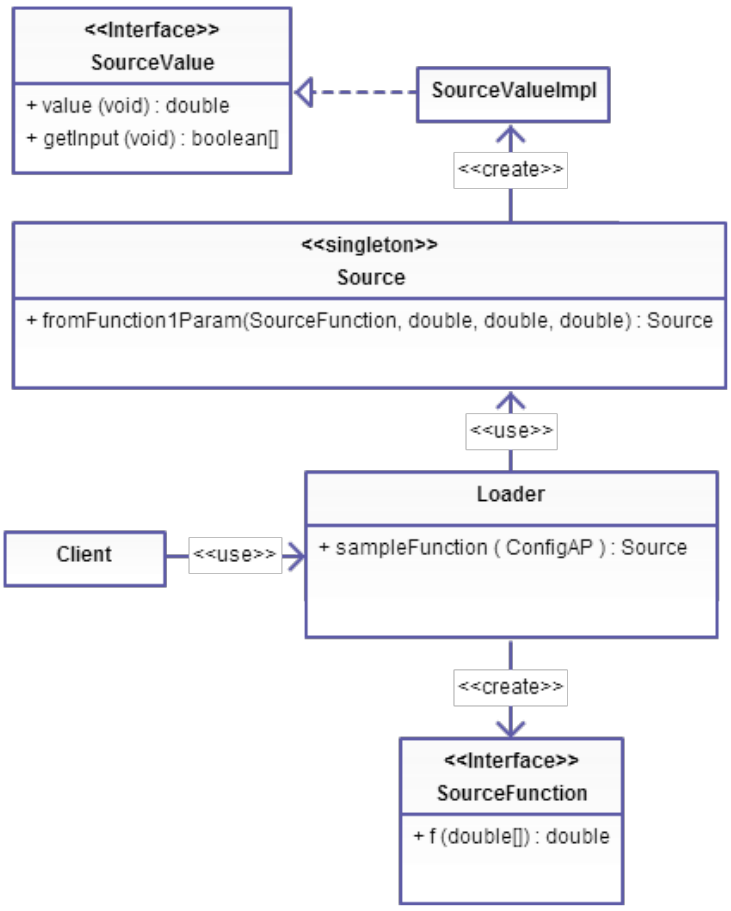
Aby byl program schopen se vstupní funkcí pracovat, je potřeba ji reprezentovat jako množinu bodů. Počet bodů, na které bude tato funkce rozložena, hraje při hledání řešení velkou roli. Pokud bude těchto bodů málo, tak program sice poběží rychleji, ale reprezentace vstupní funkce bude velice nepřesná. V opačném případě, pokud bodů bude moc, bude reprezentace sice přesnější, ale výpočet řešení bude výpočetně náročnější. Z tohoto důvodu je potřeba volit počet bodů s uvážením. Celý tento proces převodu spojité funkce na množinu bodů se nazývá vzorkování. Na obrázku 16 jsou zobrazeny dvě stejné funkce již po vzorkování, kde každá z nich je vzorkována s jiným počtem bodů. Lze vidět, že v případě malého počtu bodů se ztratí informace o tvaru původní funkce. Tento úkol mají na starost následující třídy a rozhraní, které zde budou blíže popsány.

Jeden z těchto vzorkovaných bodů, je v programu reprezentován rozhraním *SourceValue*. Jeho implementaci představuje třída *Source*, která je vytvářena pomocí metody s názvem *fromFunction1Param*. Tato metoda je kritická pro správnou interpretaci vstupní funkce jako množiny bodů. Pro její zavolání je nutné definovat minimální a maximální hranici, krok vzorkování a implementaci rozhraní *SourceFunction*. Maximální a minimální hranice říká, odkud začít vzorkovat a kde vzorkování ukončit. Krok definuje, na kolik bodů bude funkce rozložena. Rozhraní *SourceFunction* poskytuje metody, které jsou schopné vrátit hodnotu vstupní funkce v určitém bodě. Vstupní funkci lze díky knihovně *JEval* definovat přímo předpisem, protože tato knihovna je schopna interpretovat a vypočítat vstupní matematický výraz zadaný řetězcem.



Obrázek 16: Důležitost počtu vzorkovaných bodů při vzorkování funkce

Samotný proces tvorby množiny bodů z funkce o jedné proměnné probíhá v cyklu od minimální hranice vzorkování až po maximální hranici vzorkování s konstantním krokem. Aktuální hodnotu vstupní proměnné v cyklu vždy představuje pomocná proměnná, která se po každém průchodu cyklem zvětší o hodnotu kroku. V samotném těle cyklu se pak tato pomocná proměnná optimalizuje tak, aby nemohla nabývat nulové hodnoty. Pokud by totiž vstupní funkce ve jmenovateli zlomku obsahovala proměnnou, nešla by výstupní hodnota vypočítat, protože při výpočtu výrazu by program místo této proměnné dosadil nulu, kterou nelze dělit. Po optimalizaci pomocné proměnné nastává samotný výpočet bodu. Ten probíhá zavoláním metody *f*. Následně je tato vypočtená hodnota spolu s odpovídající optimalizovanou proměnnou obalena objektem třídy *SourceValueImpl*. Tento objekt je pak přidán do výstupní kolekce. Pro co nejjednodušší použití této funkcionality je vytvořena statická metoda *Loader.sampleFunction*, která se o celý tento proces postará. Metoda má pouze jeden parametr a tím je objekt reprezentující konfiguraci programu. Vše ostatní už metoda obstará sama. Na obrázku 17 je zobrazen diagram, který reprezentuje strukturu tříd zodpovědnou za převod vstupní funkce na množinu bodů.



Obrázek 17: Třídní diagram funkcionality pro vzorkování

5.4 Obecná funkční množina GFS

Pro chod *AP* je nutné mít definovanou obecnou funkční množinu. Prvky této množiny reprezentují objekty, ze kterých se může výsledné řešení skládat. V programu každá taková funkce musí dědit z abstraktní třídy *GFunction*. Tato třída představuje základní rozhraní pro práci s jakýmkoliv typem funkce v *GFS* množině. Také definuje abstraktní metody, které každá rozšiřující třída musí implementovat. Mezi tyto metody patří následující:

- **double f(double ... args)** - Jako parametr je nutno zadat vstupní hodnoty proměnných. Metoda musí vrátit hodnotu funkce.
- **int argsCount()** - Musí vrátit počet argumentů funkce.
- **GFunction clone()** - Musí vrátit nový objekt se stejnou konfigurací, jako ten na kterém je volána.

GFunction obsahuje konstruktor s jedním parametrem, který musí všechny rozšiřující třídy v jejich konstruktoru zavolat. Tento parametr říká, jak má být funkce převedena do textové reprezentace. Třída je navrhnuta tak obecně, aby zvládla reprezentovat funkce o žádné, jedné nebo více proměnných. V programu jsem implementoval pouze funkce od nuly do dvou proměnných, kdy funkce se stejným počtem parametrů dědí od stejné třídy, která nastavuje jejich základní vlastnosti. Není však žádný problém tyto třídy rozšířit a implementovat tak jiné uživatelské funkce. Všechny momentálně implementované funkce jsou zobrazeny v tabulce 8. Ukázka implementace takové funkce je zobrazena ve výpisu kódu 2.

0 proměnných	**1 proměnná**	**2 proměnné**
Konstanta	Absolutní hodnota	Dělení
	Cosinus	Sčítání
	Sinus	Odečítání
	Tangens	Násobení
	Log 10	
	Negace	

Tabulka 8: Implementované funkce v GFS

```
public class CosFunction extends OneParamFunction {
    public CosFunction() {
        super("cos(%s)");
    }

    @Override
    protected double f(double... args) {
        return Math.cos(args[0]);
    }
}
```

Výpis 2: Implementace funkce cos

Protože je potřeba, aby v *GFS* byla zahrnuta i vstupní proměnná (například x) existuje speciální typ funkce, který ji reprezentuje. Jedná se o třídu *VariableFunction*. Pro její vytvoření je potřeba určit, pro kterou ze vstupních proměnných se objekt vztahuje. Toho je docíleno dalším parametrem v konstruktoru, který definuje, jaký prvek z pole (jeho index) předaného v metodě *f* bude odpovídat hodnotě proměnné. Platí zde, že pro jednu vstupní proměnnou může existovat v *GFS* pouze jedna instance *VariableFunction*. Pokud tedy bude definovaný problém o dimenzi 2, tak budou v *GFS* dvě instance třídy *VariableFunction*. První pro proměnnou X a druhá pro proměnnou Y.

Samotnou *GFS* reprezentuje třída *GFSet*. V programu existuje pouze jedna instance této třídy, vychází totiž z návrhového vzoru *Singleton*. Instance v sobě obsahuje kolekci funkcí, které mají být použity pro evoluci. Po vytvoření instance je tato kolekce prázdná a jednotlivé funkce do ní lze přidávat metodou *addFunction*. Každá z takto přidaných funkcí má v kolekci svůj unikátní index, který je použit při samotné evoluci. Třída obsahuje další pomocné metody, díky kterým je možné s třídou lépe pracovat [28]:

- **int count()** - Vrátí počet funkcí v množině.
- **int maxParamCount()** - Vrátí, kolik nejvíce proměnných mají funkce v množině.
- **GFunction getFunctionAtIndex(int index)** - Vrátí funkci z množiny, která se nachází na indexu předaném v parametru funkce.

5.5 Specimen

Specimen definuje, jak má vypadat jedinec z pohledu struktury jeho parametrů. Při určitých aplikacích evolučních algoritmů se může stát, že není žádoucí, aby určitý parametr nabýval všech reálných hodnot. Právě tento problém řeší specimen. Je to v podstatě šablona na tvorbu jedinců. Definuje jak má jedinec, kterého pro evoluci potřebujeme, vypadat. V programu ho reprezentuje třída *Specimen*, kde její použití je velice jednoduché. Po vytvoření instance této třídy je možno volat metodu *addArgument*, kde jejím účelem je přidat do této šablony na jedince nový parametr. Metoda má dva parametry, těmi jsou minimální a maximální hodnota, jakou parametr v jedinci může nabývat. Pokud tedy bude tato metoda zavolána například třikrát, bude mít vytvořený jedinec podle tohoto specimena tři parametry v mezích, které byly předány v parametrech této metody.

Třída obsahuje další pomocné metody jako je například metoda *args()*, která vrátí celkový počet parametrů definovaných v tomto vzoru. Pro práci se také hodí metody *minForArg(int)* a *maxForArg(int)*, které jsou schopny vrátit minimální, případně maximální hodnotu parametru, kterou můžou nabývat.

Často se však stává, že je nutno, aby všechny parametry měly stejnou maximální i minimální hodnotu. Z tohoto důvodu byla vytvořena pomocná statická metoda *createSpecimen* ve třídě *Specimen*, která automaticky vytvoří instanci specimena s určitým počtem parametrů, kde minimální a maximální hodnota parametru je pro všechny parametry stejná.

5.6 Jedinec

Jedinec je reprezentován abstraktní třídou s názvem *Individual*. Tato třída definuje základní rozhraní pro práci s jakýmkoliv jedincem. Každý jedinec z ní musí dědit a implementovat následující metody:

- **double getValueAt(int index)** – Vrátí hodnotu prvku jedince na určitém indexu.
- **double[] getValues()** – Vrátí všechny prvky jedince.
- **void setValues(double[] values)** - Nastaví všechny prvky jedince na základě argumentu této funkce.
- **double getCostValue()** – Vrátí vypočtenou *CV* jedince.
- **void update()** - Vytvoří výraz na základě prvků jedince, který reprezentuje prvek z domény možných řešení problému.

- **Expression getExpression()** - Vrátí výraz z domény možných řešení problému, který reprezentuje tohoto jedince.

Třída poskytuje statické metody pro zjednodušení práce s jedinci. Jsou to například metody pro jejich vytváření a operace nad nimi. Pro vytváření kolekcí jedinců lze použít pomocnou třídu *IndividualGenerator*. Obsahuje statické metody pro generování kolekcí jedinců. Jako parametry je zde potřeba zadat požadovaný počet jedinců v kolekci, specimen objekt, který definuje kolik hodnot a v jakém rozsahu muže jedinec mít a objekt implementující rozhraní *Generator*, který slouží jako náhodný generátor čísel. Výstupem z těchto metod jsou následně kolekce jedinců s již náhodně přednastavenými hodnotami podle definice specimena.

5.7 Interpretace výrazu

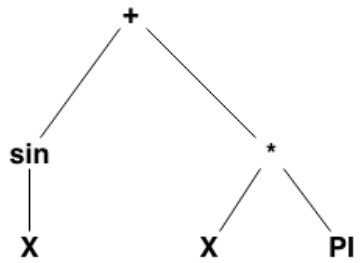
Nejdůležitější částí celého *AP* je proces převodu jedince z domény reálných hodnot na výraz složený z objektů obsažených v *GFS*. Díky tomuto mechanismu je možné pro evoluci využít jakýkoliv *EA*, protože jedinci, kteří díky němu vzniknou, můžou být jednoduše převedeni na objekty složené z prvků *GFS*.

5.7.1 Reprezentace výrazu

Aby bylo možné převod realizovat, je potřeba definovat nějakou strukturu, která bude tento převedený výraz do množiny objektu z *GFS* reprezentovat. K tomuto účelu slouží rozhraní *Expression*. Při převodu jedince se pro něho vytvoří instance na základě tohoto rozhraní, které poskytuje tyto metody:

- **double interpret(double ... input)** – Vrátí výsledek výrazu na základě pole vstupních proměnných v argumentu *input*.
- **String interpretDefinition()** – Vrátí interpretaci výrazu v textové podobě.
- **int countOfVariables()** - Vrátí počet proměnných ve výrazu.

Toto rozhraní implementuje třída *Function*, která představuje stromovou strukturu objektů z *GFS*. Ukázka, jak tato stromová struktura může vypadat, je zobrazeno na obrázku 18. Pro vytvoření její instance je potřeba definovat, jakou funkci z *GFS* má reprezentovat a jaké výrazy reprezentují parametry této funkce. Na základě těchto informací je následně metoda interpret schopna dosadit tyto parametry do funkce, kterou reprezentuje a vrátit výsledek. Třída tedy zastupuje funkci určitého objektu z *GFS* a zároveň v sobě obsahuje stromovou strukturu, do které jsou poskládány jednotlivé parametry těchto zastupovaných objektů z *GFS*.



Obrázek 18: Stromová struktura tříd Function

Celá operace interpretace je rekurzivní proces, protože metoda *interpret* musí zavolat metodu *interpret* na každý parametr funkce, který je v konstruktoru předán a tyto parametry musí zase pro všechny své parametry zavolat metodu *interpret*. Rekurze končí, jakmile výraz nemá žádné parametry. Kód pro tuto funkcionalitu je vypsán ve výpisu kódu 3. Objekt *params*, který je v tomto výpisu kódu využit, reprezentuje argumenty výrazů funkce, které jsou předány v konstruktoru.

```
public double interpret(double... input) {
    if (params.length == 0) {
        return func.func(input);
    } else {
        double[] semires = new double[params.length];
        for (int i = 0; i < params.length; i++) {
            Expression actual = params[i];
            semires[i] = actual.interpret(input);
        }
        return func.func(semires);
    }
}
```

Výpis 3: Interpretace výrazu

5.7.2 Převod jedince

Po definování, jak struktura reprezentace výrazu jedince funguje, je potřeba použít mechanismus, který na základě hodnot prvků v jedinci tuto strukturu vytvoří. Vytvoření této struktury probíhá pomocí volání metody *createExpression* ve třídě *FunctionInterpret*. Tato metoda pro svůj úkol potřebuje pouze instanci jedince, pro kterého má být výraz vytvořen. Po jeho vytvoření vrátí instanci objektu implementující rozhraní *Expression*, který představuje kořenový element celé struktury. Samotná metoda *createExpression* je však pouze pro uživatele zjednodušené volání rekurzivní funkce *recursiveExpressionComposit*, která má na starost samotnou tvorbu struktury výrazu z hodnot jedince. Pro její volání je potřeba definovat parametry potřebné pro rekurzi. Těmi jsou aktuální index z pole hodnot jedince. Dále pole *boolean* hodnot, které říká, jaké prvky pole jedince už byly použity. V neposlední řade je to také instance jedince, pro kterého se struktura vytváří a konečně instance třídy *GFunction*, která reprezentuje funkci z *GFS* na místě aktuálního indexu.

Protože je metoda rekurzivní, tak v první fázi kontroluje, zda ještě může vykonávat svou činnost. Tato situace nenastane, pokud v jednom z předchozích rekurzivních běhů byl použit předposlední volný index. Za těchto okolností musí vrátit obalenou aktuální funkci třídou *Function*. Pokud však aktuální funkce má alespoň jeden parametr, nastává problém. Žádný další index není již volný a není tedy za parametr možné dosadit hodnotu. V této situaci musí metoda místo aktuální funkce obalit třídou *Function* pseudonáhodnou jinou funkci s nulovým počtem argumentů.

Pokud je v jedinci dostatečné množství volných indexů, pak lze průběh metody rozdělit na dvě situace. První představuje situaci, kdy aktuální funkce má dostatečný počet argumentů ve formě volných indexů. Druhá představuje situaci, kdy aktuální funkce nemá k dispozici dostatečný počet argumentů ve formě volných indexů. V prvním případě se pracuje s aktuální funkcí, protože pro ni existují argumenty. V druhém případě se aktuální funkce nahradí jinou funkcí tak, aby vycházel počet argumentů. Následující zpracování je pro obě situace totožné. Metoda najde volné indexy pro všechny parametry funkce a následně pro každý tento parametr zavolá sama sebe. Toto volání pro každý parametr vrátí *Expression*. Výsledkem takového volání pro každý parametr je tedy pole *Expression[]*, které je následně spolu s aktuální funkcí obaleno třídou *Function* a vráceno z rekurze. Toto řešení tedy zabraňuje, aby vznikl patologický výraz. Tuto funkcionalitu reprezentuje kód ve výpisu kódu 4.

```
private Expression recursiveExpressionComposit(Boolean[] visitedData,
        Individual ind, int indexPosition, GFunction fun) {
    if (!hasAlmostOneFreeIndex(visitedData)) {
        if (fun.argsCount() == 0) {
            return new Function(fun);
        } else {
            return new Function(set.getRandomFunction(0, indexPosition));
        }
    }
        int maxArgs = maxArgumentsForUse(visitedData);
    if (fun.argsCount() <= maxArgs) {
            int argsCount = fun.argsCount();
        int freeIndexes[] = new int[argsCount];
        Expression[] newexp = new Expression[argsCount];
        int actualIndex = 0;
        for (int i = indexPosition; i < ind.size(); i++) {
            if (visitedData[i] == false && actualIndex < argsCount) {
                freeIndexes[actualIndex] = i;
                visitedData[i] = true;
                actualIndex++;
            }
        }
            for (int i = 0; i < argsCount; i++) {
            newexp[i] = recursiveExpressionComposit(visitedData, ind,
                    freeIndexes[i], set.getFunctionAtIndex((int) Math
                            .round(ind.getValueAt(freeIndexes[i]))));
        }
        return new Function(fun, newexp);
    } else {
        GFunction f = set.getRandomFunction(maxArgs, indexPosition);
```

```
            int argsCount = fun.argsCount();
            int freeIndexes[] = new int[argsCount];
            Expression[] newexp = new Expression[argsCount];
            int actualIndex = 0;
            for (int i = indexPosition; i < ind.size(); i++) {
                if (visitedData[i] == false && actualIndex < argsCount) {
                    freeIndexes[actualIndex] = i;
                    visitedData[i] = true;
                    actualIndex++;
                }
            }
            for (int i = 0; i < argsCount; i++) {
                newexp[i] = recursiveExpressionComposit(visitedData, ind,
                        freeIndexes[i], set.getFunctionAtIndex((int) Math
                                .round(ind.getValueAt(freeIndexes[i]))));
            }
            return new Function(f, newexp);
        }
    }
```

Výpis 4: Metoda vytvářející výslednou strukturu na základě jedince

5.8 Posílené hledání

V programu je posílené hledání reprezentováno třídou *ReinforcedSearchFunction*, která implementuje rozhrani *GFunction*. Pro vytvoření instance funkce pro posílené hledání je potřeba v konstruktoru předat výraz *Expression*, který reprezentuje interpretovaný výraz jedince, který má být pro posílené hledání použit. Samotné vytvoření instance *ReinforcedSearchFunction* však nestačí. Je potřeba tuto instanci předat implementaci *GFS*. To je realizováno metodou *reinforcedSearch*, která jako parametr vyžaduje implementaci rozhraní *GFunction*, takže také *ReinforcedSearchFunction*. Zavoláním této metody se spustí mechanismus vypsaný ve výpisu kódu 5.

```
    public void reinforcedSearch(GFunction function) {
        if (!hasReinforcedSearch) {
            functions.add(function);
            hasReinforcedSearch = true;
        } else {
            functions.remove(functions.size() - 1);
            functions.add(function);
        }
    }
```

Výpis 5: Mechanismus zavedení posíleného hledání do GFS

Třída *GFSet* si v sobě udržuje stav, zda je posílené hledání zapnuto nebo vypnuto. Pokud je vypnuto, tak se *GFunction* předaná v parametru přidá do aktuálních funkcí v *GFS* a změní se stav na zapnuto. Pokud je již zapnuto, tak se poslední funkce přidaná do *GFS* nahradí funkcí předanou v parametru. Tím se stará funkce zamění za novou.

Protože zapnutím posíleného hledání se rozšíří počet funkci v *GFS*, je potřeba o této změně informovat objekt specimen, který se používá pro vytváření jedinců. K tomuto

úkonu slouží metoda *enableReinforcement*. Aby mohla být funkce z rozšířeného hledání někdy použita, musí se zvětšit maximální hodnota indexu v jedinci, což řeší funkce zobrazena ve výpisu kódu 6.

```
public void enableReinforcement() {
    if (!isReinforcement) {
        this.isReinforcement = true;
        for (int i = 0; i < data.size(); i++) {
            SpecimenItem si = data.get(i);
            si.max += 1;
        }
    }
}
```

Výpis 6: Povolení posíleného hledání ve vzorovém jedinci

Princip činnosti této metody spočívá v tom, že pokud ještě nebylo zapnuto posílené hledání ve specimen objektu, tak u všech parametrů zvětši maximální hodnotu, kterou můžou nabývat o jedna. Tím je docíleno, že se generují jedinci, kteří v sobě obsahují funkci přidanou v posíleném hledání.

5.9 Výpočet CV

CV udává, jak je jedinec v populaci kvalitní vzhledem k řešení definovaného problému. V programu je evoluce naprogramována jako maximalizační úloha, kde nejhorší jedinec má *CV* rovno mínus nekonečnu a nejlepší jedinec jí má rovno 0. Pro výpočet *CV* bylo vytvořeno rozhraní *CostValue*, které ve svém předpise obsahuje následující metody:

- **double costValue()** – Vrátí hodnotu *CV* pro daného jedince.
- **void setIndividual(Individual ind)** – Nastaví jedince, pro kterého má být *CV* vypočítána.

Implementaci tohoto rozhraní reprezentuje třída *APCostValue*. Pro vytvoření instance z této třídy, je potřeba v konstruktoru předat instanci jedince, pro kterého má být *CV* vypočtena a instanci třídy *Source*, která reprezentuje vstupní funkci. Výpočet pak probíhá způsobem, že se pro každý vzorkovaný bod vstupní funkce vypočítá rozdíl mezi funkční hodnotou v tomto bodě a hodnotou interpretovanou z výrazu jedince. Všechny tyto vypočítané hodnoty se sečtou a funkce vrátí jejich zápornou hodnotu.

5.10 Implementace evolučních algoritmů

EA reprezentují evoluční jádro celého *AP*. Určují, jakým způsobem budou vytváření noví jedinci v populaci. Jejich volba a nastavení má přímý vliv na výpočetní rychlost a kvalitu výsledku řešeného problému. Uživatel programu si může včetně jejich parametrů explicitně určit, jaký algoritmus má být pro evoluci použit.

V programu *EA* implementují rozhraní *EvolutionAlgorithm*, které definuje základní operace, které nad nimi lze vykonávat. Rozhraní se skládá z následujících metod:

- **List<Individual> execute()** – Vrátí novou generaci populace v rámci *EA*.
- **boolean isNextIteration()** – Zjistí, zda algoritmus může vytvořit další generaci populace. Tento stav nastane, pokud nejsou naplněny ukončovací parametry algoritmu.
- **void reset()** - Nastaví algoritmus do výchozího stavu. Tuto metodu je nutné volat před každou evolucí.

Cílem tohoto přístupu je poskytnou abstrakci nad *EA*, se kterou můžou pracovat vyšší vrstvy. Tyto vrstvy se starají o jejich správné použití v rámci evolučního cyklu, volají tedy metody tohoto rozhraní.

5.10.1 Evoluční strategie

Implementace algoritmů evolučních strategií se nachází v balíčku *evolutionalgorithm.evstrategy*. Jsou implementovány celkem tři typy těchto strategií. Pro vytvoření jejich instancí existují určité společné rysy. U všech je potřeba mimo jiné definovat tyto parametry:

- Specimen pro vytváření nových jedinců
- Generátor pro generování náhodných čísel
- Počet iterací algoritmu
- Směrodatnou odchylku pro Gaussovo normální rozdělení
- Cílovou *CV*, ve které se algoritmus zastaví

Tyto algoritmy mají také stejné implementace metod *reset* a *isNextIteration* z rozhraní *EvolutionAlgorithm*. Funkce *reset* resetuje čítač aktuální iterace a znovu vytvoří náhodnou počáteční populaci. Funkce *isNextIteration* je pravdivá pouze v případě, že nejlepší *CV* jedince v populaci nepřesáhne cílovou *CV* předanou v konstruktoru a aktuální iterace je menší než maximální počet iterací.

5.10.1.1 Dvoučlenné Tento algoritmus reprezentuje třída *ESOnePlusOneAlgorithm*. Jedná se o nejtriviálnější verzi *ES*. Tento algoritmus nepotřebuje definovat další parametry, než ty, které jsou společné pro všechny algoritmy *ES*. Implementace metody execute je následující:

```
public List<Individual> execute() {
    Individual  neighbour = getNeighbour(res);
    if  (neighbour.getCostValue() > res.getCostValue()) {
        res = neighbour;
    }
    actualIter ++;
    return Arrays.asList(res);
}
```

Výpis 7: Implementace metody execute pro algoritmus dvoučlenné ES

Kritická je zde metoda *getNeighbour*. Je zodpovědná za vytvoření nového jedince na základě aktuálního a to pomocí Gaussova normálního rozdělení. Princip její činnosti spočívá v tom, že se v cyklu pro každý prvek jedince vypočítá podle Gaussova normálního rozdělení přírůstek a jeho polarizace, která určí, zda tento přírůstek je kladný nebo záporný. Takto získaný přírůstek se následně přičte k aktuálnímu prvku jedince a vznikne tak nová hodnota. Tímto způsobem vzniká mutace jedince. Následně se testuje, zda aktuální prvek jedince vyhovuje předpisu specimen. Pokud se hodnota prvku nachází mimo povolené hranice definované specimenem, tak se celý cyklus opakuje a hledá se takový přírůstek, který vyhovuje tomuto požadavku.

Jedna iterace tohoto algoritmu se tedy skládá z vytvoření sousedícího jedince, který je porovnán se stávajícím jedincem. Pokud je tento nový jedinec lepší než jedinec starý, je starý jedinec nahrazen tímto novým jedincem. Následně se provede inkrementace čítače iterací a vrátí se takto získaný jedinec.

5.10.1.2 Vícečlenné Tento algoritmus reprezentuje třída *ESMultiAlgorithm*. Pro vytvoření instance tohoto algoritmu je potřeba kromě společných parametrů ještě definovat velikost populace. Oproti dvoučlennému *ES* se zde tedy vyskytuje již populace jedinců. Implementace metody *execute* je následující:

```
public List<Individual> execute() {
    List<Individual> newRes = new ArrayList<>();
        for ( Individual i : res) {
         Individual neighbour = getNeighbour(i);
         if (neighbour.getCostValue() > i.getCostValue()) {
            newRes.add(neighbour);
        } else {
            newRes.add(i);
        }
    }
    List<Individual> union = new ArrayList<>();
    union.addAll(newRes);
    union.addAll(res);
    Collections.sort(union, new IndividualComparator());
        List<Individual> newPop = new ArrayList<>();
    int count = res.size();
    for (int i = 0; i < count; i++) {
        newPop.add(union.get(i));
    }
    this.res = newPop;
     actualIter++;
    return newPop;
}
```

Výpis 8: Implementace metody execute pro algoritmus vicečlenné ES

Pro každého jedince z počáteční populace se pomocí metody *getNeighbour* vygeneruje sousední jedinec. Tato metoda pracuje stejně jako v případě dvoučlenné *ES*. Tento vygenerovaný jedinec se porovná s aktuálním jedincem v cyklu, a pokud je kvalitnější, tak je přidán do pomocné kolekce *newRes*. V opačném případě je do pomocné kolekce

přidán aktuální jedinec v cyklu. Po tomto cyklu je provedeno sjednocení kolekce *newRes* s aktuální populací. Výsledek tohoto sjednocení je seřazen od nejlepšího jedince po nejhoršího. Následně je z tohoto sjednocení vybráno prvních X nejlepších jedinců (záleží na velikosti populace) ze kterých je vytvořena nová populace, která nahradí tu stávající. Pak se provede inkrementace čítače iterací a vrátí se takto získaná populace jako návratová hodnota metody.

5.10.1.3 Rekombinační Tento algoritmus reprezentuje třída *ESRecombinantAlgorithm*. Pro vytvoření instance tohoto algoritmu je potřeba kromě společných parametrů ještě definovat tyto parametry:

- Velikost populace
- Kolik rodičů se má použít pro rekombinaci
- Zdali se jedná o diskrétní rekombinaci (pokud se nejedná, tak je použita průměrová metoda)

Metoda *execute* je v tomto případě totožná s vícečlennou *ES*. Rozdíl je zde však v metodě pro vytvoření mutace jedince, tedy *getNeighbour*. Ten spočívá v tom, že se přírůstek nepřičítá přímo k prvku jedince, ale k prvku, který vznikne po rekombinaci. Pro výpočet rekombinantu slouží metoda *recombinant*, do které je nutno předat instanci zdrojového jedince. Při samotné rekombinaci se nejprve najde potřebný počet rodičů, kteří budou v rekombinaci použiti. Tato činnost se odehrává v cyklu, který se opakuje, dokud není nalezen dostatečný počet rodičů. Je tedy nutné mít velikost populace větší než je počet hledaných rodičů pro rekombinaci. Jedinec totiž nemůže být v rámci jedné rekombinace použit vice jak jednou. Následně se výpočet větví na dvě části. To, která se uplatní, závisí na tom, zda bylo v konstruktoru předáno, že se jedná o diskrétní rekombinaci. Tato situace vypadá v kódu následovně:

```
if (this.isDiscrete) {
    for (int i = 0; i < res.length; i++) {
        int index = (int) gen.randomInRange(0, recombinationCount);
        res[i] = selected.get(index).getValueAt(i);
    }
} else {
    for (int i = 0; i < res.length; i++) {
        double actual = 0.0;
        for (Individual j : selected) {
            actual += j.getValueAt(i);
        }
        res[i] = actual / (double) selected.size();
    }
}
```

Výpis 9: Implementace metody execute pro algoritmus rekombinační ES

Výstupní rekombinant reprezentuje pole *double* čísel s názvem *res*. Jeho velikost je dána počtem prvků v jedinci (dimenze). V případě diskrétní rekombinace probíhá tvorba výstupního rekombinantu tak, že se pro každý prvek rekombinantu nejprve náhodně vybere jeden rodič z již vybraných rodičů a na místo aktuálního prvku pole se uloží jeho hodnota na stejném prvku, jako je aktuální prvek. Zpracování pro průměrovou rekombinaci probíhá trochu jiným způsobem. Pro každý prvek rekombinantu se vypočítá průměr odpovídajících prvků ve všech vybraných rodičích pro rekombinaci. Takto získané hodnoty reprezentují výsledný rekombinant.

5.10.2 Diferenciální evoluce

Implementace *DE* se nachází v balíčku *evolutionalgorithm.de*, konkrétně se jedná o třídu *DiferencialAlgorithm*. Pro vytvoření instance tohoto algoritmu je potřeba definovat parametry, které byly popsány v kapitole 4.2.

Metoda *reset* má totožnou implementaci, jako v případě algoritmu *ES*. Mírně se však liší implementace metody *isNextIteration*. Ta vrátí pravdivou hodnotu pouze v případě, že nebylo dosaženo maximálního počtu iterací. Pro vytváření *noisy* a *trial* vektorů obsahuje třída pomocné metody, které tento úkol plní. K tomu, aby pro nějakého jedince mohl být vytvořen *noisy* vektor, je potřeba nejprve náhodně vybrat tři další jedince. K tomuto úkolu slouží metoda *find3ByRand*, která jako parametry vyžaduje aktuálního jedince a kolekci celé populace. Tato metoda následně náhodně vybere z populace tři jedince, kteří jsou od sebe různí a vrátí jejich kolekci. Po vybrání těchto jedinců je již možné vytvořit *noisy* vektor. Tuto funkcionalitu v sobě obsahuje metoda *createNoisy*, které je v parametru předána právě již zmíněná kolekce náhodně vybraných jedinců. Samotný *trial* vektor je poté vytvořen z aktuálního jedince a již vytvořeného *noisy* vektoru. Vytvoření *trial* vektoru má na starost metoda *createTrial*. Jako parametry vyžaduje instanci aktuálního jedince, pro kterého má být *trial* vytvořen a již zmíněný *noisy* vektor.

V některých případech se pří vytváření *trial* vektoru může stát, že některé z prvků vektoru neodpovídají požadovaným mezím, které jsou definovány specimenem. Je tedy potřeba vždy zkontrolovat, zda vektor odpovídá požadovaným omezením a případné problémy opravit. K tomuto účelu slouží metoda *correctValues*, která je zobrazena ve výpisu kódu 10.

```
private Individual correctValues(Individual individual) {
    double[] data = individual.getValues();
    double[] newData = new double[data.length];
    for (int j = 0; j < data.length; j++) {
        double val = data[j];
        if (val < specimen.minForArg(j) || val > specimen.maxForArg(j)) {
            newData[j] = gen.randomInRange(specimen.minForArg(j),
                specimen.maxForArg(j));
        } else {
            newData[j] = data[j];
        }
    }
    return Individual.createIndividual(newData);
}
```

Výpis 10: Metoda correctValues

Tato metoda projde všechny prvky vektoru a zkontroluje, zda se prvek nachází v povolených mezích. V případě, že se nachází mimo mez, je tento prvek vektoru nahrazen náhodným číslem z povoleného intervalu definovaného specimenem. Implementace metody *execute* pracuje v cyklu přes celou aktuální populaci. Pro aktuálního jedince z populace je vždy vypočítán *trial* vektor, který je pomocí metody *correctValues* opraven. Následně je jeho kvalita porovnaná s aktuálním jedincem, kde ten lepší z nich bude součástí nové populace.

5.10.3 SOMA

Implementace *SOMA* algoritmů se nachází v balíčku *evolutionalgorithm.soma*. Celkově jsou implementovány tyto strategie:

- Všichni k jednomu
- Všichni ke všem
- Adaptivně všichni ke všem
- Všichni k jednomu náhodně

Všechny tyto třídy rozšiřují abstraktní třídu *BaseSoma*. Tato abstraktní třída obsahuje konstruktor, který definuje povinné atributy pro vytváření instancí z ní odvozených. Pro vytvoření instance jakékoliv strategie je potřeba definovat tyto atributy:

- Instanci specimen objektu pro tvorbu jedinců
- Instanci generátoru náhodných čísel
- Velikost populace
- Počet iterací
- Délku cesty
- Krok
- Perturbaci

Třída *BaseSoma* obsahuje společné metody pro všechny strategie algoritmů. Některé třídy, z ní odvozené, je pro svou činnost používají. Jedná se hlavně o metody *createPRTVector*, *correctIndividual*, *findBest* a *migrateIndividual*. Metoda *createPRTVector* je zodpovědná za vytvoření vektoru pro perturbaci. Ten má stejný počet prvků jako jedinec s tím rozdílem, že každý prvek je tvořen buď číslem nula, nebo jedna. O tom, které z těchto čísel to ve skutečnosti bude, rozhoduje náhoda. Metoda *correctIndividual* zde má stejnou funkci jako

v případě *DE*, tedy zkontroluje a případně opraví prvky, které překračují povolenou mez danou specimenem. Metoda *findBest* slouží pouze pro vyhledání nejkvalitnějšího řešení z populace, která je předána v parametru metody. Metoda *migrateIndividual* slouží k migraci jedince směrem k jinému jedinci. Její implementace je zobrazena ve výpisu kódu 11.

```
protected Individual migrateIndividual(Individual from, Individual to,
        double t, double[] prtVector) {
    double[] dataFrom = from.getValues();
    double[] dataTo = DoubleHelper.sub(to.getValues(), from.getValues());
    double[] dataNew = new double[dataFrom.length];
    for (int i = 0; i < dataNew.length; i++) {
        dataNew[i] = dataFrom[i] + dataTo[i] * t * prtVector[i];
    }
    correctIndividual(dataNew);
    Individual ret = Individual.createIndividual(dataNew);
    ret.update();
    return ret;
}
```

Výpis 11: Metoda migrateIndividual algoritmu SOMA

Tato operace pro svou funkci mimo zdrojového a cílového jedince využívá parametr *t* a perturbační vektor. Parametr *t* určuje, jak hodně se má zdrojový jedinec posunout. Perturbační vektor říká, kterým směrem. Výpočet posunutí se provádí pro každý prvek jedince v cyklu *for*. Jedná se o rozšířenou parametrickou rovnici přímky, kde zdrojový jedinec představuje počáteční bod, rozdíl mezi cílovým a zdrojovým jedincem představuje směrový vektor, parametr přímky představuje parametr *t* a perturbační vektor říká, který prvek jedince bude zafixován. Výsledek je pak pomocí metody *correctIndividual* opraven. Z něho je následně vytvořena nová instance jedince, která je také návratovou hodnotou metody.

Všechny strategie mají metody *isNextIteration* a *reset* implementovány stejným způsobem. Pomocí metody *reset* se vynuluje čítač iterací a náhodně se vygeneruje počáteční populace. Metoda *isNextIteration* vrátí logicky pravdivou hodnotu pouze v případě, že aktuální iterace nepřesáhla maximální počet iterací.

5.10.3.1 Všichni k jednomu

Implementaci této strategie představuje třída *SomaAllToOneAlgorithm*. Výpočet nové populace probíhá v cyklu přes všechny jedince aktuální populace. Pokud je aktuální jedinec zároveň nejlepším jedincem populace, tak je průchod cyklem ukončen a pokračuje se dalším jedincem. Níže uvedený kód (výpis kódu 12) zobrazuje, jak je implementována migrace jednoho jedince z populace.

```
List<Individual> lineMigration = new ArrayList<>();
double[] prtVector = createPRTVector();
double distance = Individual.distance(actual, leader);
for (double lineIter = 0; lineIter < (this.pathLength * distance); lineIter += this.step) {
    lineMigration.add(migrateIndividual(actual, leader, lineIter, prtVector));
}
newPop.add(findBest(lineMigration));
```

Výpis 12: Migrace v algoritmu SOMA AllToOne

Pro ukládání potencionálních nových již zmigrovaných jedinců je použita kolekce *lineMigration*. Následně se vypočítá perturbační vektor a vzdálenost mezi aktuálním jedincem a nejlepším jedincem v populaci. V cyklu se následně vygenerují zmigrovaní jedinci, kteří sou přidáni do kolekce *lineMigration*. Toto generování je závislé na hodnotě *lineIter*, která reprezentuje parametr parametrické rovnice přímky. S každým průchodem cyklu se tato hodnota zvětšuje o v konstruktoru definovaný krok. Do nové populace se následně přidá nejlepší jedinec z kolekce *lineMigration*.

5.10.3.2 Všichni ke všem

Implementaci této strategie představuje třída *SomaAllToAllAlgorithm*. Kód se na rozdíl od strategie všichni k jednomu liší pouze v metodě *execute*. Následující kód ve výpisu 13 představuje migraci jednoho jedince, kde *actual* proměnná reprezentuje aktuálně vybraného jedince pro migraci. Pro migraci celé populace je tedy potřeba kód spustit tolikrát, kolik je jedinců v populaci.

```
for (int mig = 0; mig < population.size(); mig++) {
    Individual migTo = population.get(mig);
    if (migTo == actual)
        continue;
    List<Individual> lineMigration = new ArrayList<>();
    double[] prtVector = createPRTVector();
    double distance = Individual.distance(actual, migTo);
    for (double lineIter = 0; lineIter < (this.pathLength * distance); lineIter += this.step) {
        lineMigration.add(migrateIndividual(actual, migTo, lineIter, prtVector));
    }
    Individual bestInLine = findBest(lineMigration);
    population.remove(act);
    population.add(act, bestInLine);
    actual = bestInLine;
}
```

Výpis 13: Migrace konkrétního jedince v algoritmu SOMA AllToAll

V cyklu se prochází všichni jiní jedinci, než je aktuálně vybraný. Jejich výpočet migrace je následně totožný se strategií všichni k jednomu s tím rozdílem, že do kolekce *lineMigration* se pro aktuálního jedince ukládají potenciální možné migrace přes všechny jedince populace.

5.10.3.3 Adaptivně všichni ke všem

Implementaci této strategie představuje třída *SomaAllToAllAdaptiveAlgorithm*. Oproti implementaci strategie všichni ke všem, se ve své implementaci liší pouze v metodě *execute*. Kód ve výpisu 14 zobrazuje migraci jednoho jedince. Hlavní rozdíl je v tom, že aktuální jedinec se zde může měnit. Nejlepší jedinec je totiž vyhledáván průběžně, né až po zjištění všech potencionálních migrací jedince. Tento jev zobrazují poslední čtyři funkční řádky kódu ve výpisu 14.

```
for (int mig = 0; mig < population.size(); mig++) {
    Individual migTo = population.get(mig);
    if (migTo == actual)
        continue;
    List<Individual> lineMigration = new ArrayList<>();
```

```
        double[] prtVector = createPRTVector();
        double distance = Individual.distance(actual, migTo);
        for (double lineIter = 0; lineIter < (this.pathLength * distance); lineIter += this.step) {
            lineMigration.add(migrateIndividual(actual, migTo, lineIter, prtVector));
        }
        Individual bestInLine = findBest(lineMigration);
        population.remove(act);
        population.add(act, bestInLine);
        actual = bestInLine;
    }
```

Výpis 14: Migrace konkrétního jedince v algoritmu SOMA AllToAllAdaptive

5.10.3.4 Všichni k jednomu náhodně

Tuto strategii představuje třída *SomaAllToOneRandAlgorithm*. Její implementace se oproti strategii všichni k jednomu liší pouze ve stylu vyhledání jedince, ke kterému bude migrace probíhat. Ten je vybrán náhodně z aktuální populace. Následující kód ve výpisu 15 zobrazuje použitou metodu pro vyhledání tohoto jedince.

```
    private Individual findLeaderByRand(List<Individual> population,
            Individual actual) {
        Individual res = null;
        while (res == null) {
            int random = (int) gen.randomInRange(0, population.size() - 1);
            if (actual != population.get(random)) {
                res = population.get(random);
            }
        }
        return res;
    }
```

Výpis 15: Metoda pro nalezení náhodného leadera v algoritmu SOMA se strategií AllToAllRand

Metoda hledá náhodný index z povoleného rozsahu. Pokud se jedinec z populace na tomto indexu liší od aktuálního jedince, je tento jedinec výsledkem metody. Pokud aktuální jedinec je totožný s jedincem z populace na tomto indexu, tak se celý proces hledání opakuje.

5.10.4 Rojení částic

Tento algoritmus představuje třída *ParticleSwarmAlgorithm*, která se nachází v balíčku *evolutionalgorithm.pswarm*.

Algoritmus používá pro reprezentaci částic třídu *Individual*, která je rozšířena o možnost přidat do jedince vlastní atributy a tak v jeho instanci uchovat další informace specifické pro algoritmy, které ho využívají. Analytický vzor, který byl pro implementaci vlastních atributů použit, nese název *Dynamic Properties*. K tomuto účelu obsahuje třída *Individual* metody *hasAttr*, *setAttr* a *getAttr*. Metoda *hasAttr* slouží pro zjištění, zda

v jedinci je daný atribut nastaven. Metody *setAttr* a *getAttr* slouží pro nastavení a získání atributů. [29]

Nutnost uchovávat v jedinci další informace je pro algoritmus rojení částic kritická, protože je potřeba pro každého jedince uchovávat jeho vektor rychlosti, jeho doposud nejlepší pozici a informace o globální nejlepší pozici.

Implementace metody *isNextIteration* vrací logicky pravdivou hodnotu pouze v případě, že aktuální iterace je menší než maximální povolená. Metoda *reset* vygeneruje počáteční populaci částic a nastaví potřebné hodnoty do výchozího nastavení. Samotná metoda *execute* má implementaci zobrazenou ve výpisu kódu 16.

```
@Override
public List<Individual> execute() {
    migrations++;
    if (! inited ) {
        initDataIfNotYet (population);
        inited = true;
    }
    updateGBest(population);
    updatePBest(population);
    updatePopulationPosition(population);
    correctValues(population);
    actualIter ++;
    if ( actualIter == this. iterations ) {
        Individual gb = Individual . createIndividual (gBestVector);
        population.add(gb);
    }
    return population;
}
```

Výpis 16: Metoda execute v algoritmu Rojení částic

Jako první operace, která se provede, je inkrementace čítače migrací. Dále pokud ještě nebyla provedena inicializace atributů, je zavolána metoda *initDataIfNotYet*. Účelem této metody je pro každého jedince v populaci nastavit vlastní atributy pro jeho vektor rychlosti a nejlepší dosavadní řešení, kterého jedinec dosáhl. Dále tato metoda nastaví do privátních proměnných informace o globálním nejlepším řešení. Pro uložení globálního řešení je využito dvou proměnných, první představuje samotné pole hodnot jedince a druhá jeho *CV*. U jednotlivých jedinců v populaci se jejich dosavadní nejlepší řešení taktéž ukládá do dvou atributů. Pro každého jedince je vektor rychlosti na základě generátoru předaného v konstruktoru náhodně vygenerován.

Po samotné inicializaci algoritmu nastává pomocí metod *updateGBest* a *updatePBest* aktualizace nejlepších řešení. Metoda *updateGBest* aktualizuje privátní proměnné algoritmu tak, aby obsahovaly potřebné data z nejlepšího jedince z populace, která je poskytnuta jako parametr této metody. Aktualizaci nejlepších hodnot jedince má na starost metoda *updatePBest*. Díky tomu, že pří každém volání metody *execute* jsou tyto aktualizace provedeny, je zaručeno, že globální nejlepší jedinec a nejlepší hodnota jakou jedinec kdy nabýval, budou nastaveny vždy korektně.

Metoda, která je následně zodpovědná za samotnou migraci jedinců se nazývá *updatePopulationPosition*. Tato privátní metoda je volána metodou *execute* ihned po aktualizaci

nejlepších řešení a zajišťuje pohyb jedinců v prostoru možných řešení. Toho dosahuje prostřednictvím změny rychlostního vektoru. Pro každého jedince v populaci je vektor rychlosti nastaven na základě kódu zobrazeného ve výpisu kódu 17.

```
for (int d = 0; d < i.size(); d++) {
    double rand = gen.randomInRange(0.00001, 1);
    double w = wStart - ((wStart - wEnd) * (double) iterations) / (double) migrations;
    vNext[d] = w * vAct[d] + c1 * rand * (pBest[d] - pAct[d]) + c2
        * rand * (gBestVector[d] - pAct[d]);
    pNext[d] = pAct[d] + vNext[d];
}
vNext = DoubleHelper.mult(DoubleHelper.norm(vNext), vmax);
i.setAttr(ATTR_SPEED, vNext);
i.setValues(pNext);
```

Výpis 17: Výpočet rychlostního vektoru v algoritmu Rojení částic

Aktuálního jedince představuje proměnná *i*. V cyklu *for* se pro každý prvek nejprve vygeneruje náhodné číslo, které je použito pro výpočet odpovídajícího prvku vektoru. Kromě proměnných definovaných v konstruktoru algoritmu se zde ve výpočtu vyskytují další proměnné. Pole *pAct* představuje prvky aktuálního jedince. Pole *vAct* zase představuje aktuální vektor rychlosti jedince. Pro reprezentaci prvků nejlepšího nalezeného řešení, kterého konkrétní jedinec nabýval, slouží pole *pBest*. Nově vzniklý vektor rychlosti reprezentuje proměnná *vNext*. Nové hodnoty po migraci jedince představuje pole *pNext*. Všechny prvky tohoto rychlostního vektoru jsou následně pomocí statické metody *norm* třídy *DoubleHelper* normalizovány tak, aby maximální délka tohoto vektoru byla omezena. Vektor je následně uložen do vlastního atributu jedince, pro kterého byl vypočítán. Po vypočtení nových pozic jedinců se provede pomocí metody *correntValues* kontrola jejich mezí. Prvky jedince, které nevyhovují mezím specimenu, jsou upraveny tak, aby vyhovovaly.

5.10.5 Simulované žíhání

Implementace tohoto algoritmu se nachází v balíčku *evolutionalgorithm.sa*. Třída, která algoritmus představuje má název *SimulatedAnnealingAlgorithm*.

Metody *reset* a *isNextIteration* jsou zde oproti jiným algoritmům implementovány trochu odlišně. Algoritmus nepracuje totiž s populací ale jedním jedincem a teplotou. Metoda *reset* nastaví aktuální teplotu na počáteční a vygeneruje počátečního jedince, kterého zároveň zvolí jako nejlepší nalezené řešení. Metoda *isNextIteration* vrátí logicky pravdivou hodnotu pouze v případě, že je aktuální teplota větší nebo rovna konečné teplotě.

Funkci pro redukci teploty lze nastavit pomocí rozhraní *Alpha*. Toto rozhraní obsahuje metodu *f*, která jako parametr požaduje aktuální teplotu a vrátí redukovanou teplotu. Implementace tohoto rozhraní je použita v metodě *execute* právě pro redukci teploty. Jedna iterace algoritmu je implementována způsobem zobrazeným ve výpisu kódu 18.

```
@Override
public List<Individual> execute() {
    for (int i = 0; i < metropolisCount; i++) {
```

```
            Individual neighbour = getNeighbour(xZero);
            double f = neighbour.getCostValue() − xZero.getCostValue();
            if (f > 0) {
                xZero = neighbour;
                if (neighbour.getCostValue() > xStar.getCostValue()) {
                    xStar = neighbour;
                }
            } else {
                Random rand = new Random();
                double r = rand.nextGaussian();
                double e = Math.exp(−f / tActual);
                if (r < e) {
                    xZero = neighbour;
                }
            }
        }
        this.tActual = this.alpha.f(tActual);
        return Arrays.asList(this.xStar);
    }
```

Výpis 18: Medota execute v algoritmu Simulovaného žíhání

Tato metoda obsahuje cyklus metropolisova algoritmu. Počet opakování pro jednu teplotu je definován v konstruktoru. Tento algoritmus je reprezentován cyklem *for*, kde při každém průchodu se vždy na základě aktuálního jedince vypočítá jeho sousední jedinec. Takovýto výpočet má na starost metoda *getNeighbour*, která funguje stejně jako v případě algoritmu dvoučlenné *ES*. Na základě takto vypočteného jedince se vypočítá rozdíl mezi jeho a aktuálním *CV*. Pokud je tato hodnota větší než nula, jinak řečeno vypočtený sousední jedince je kvalitnější než stávající, dojde k nahrazení aktuálního jedince tímto vypočteným jedincem. Dále pokud je tento jedinec kvalitnější než nejlepší jedinec za celou existenci algoritmu, je dosavadní nejlepší jedinec nahrazen tímto jedincem. Pokud je hodnota rozdílů *CV* menší nebo rovna nule, pak je nový aktuální jedinec vyměněn pouze na základě pravděpodobnostního výpočtu. Po skončení činnosti tohoto algoritmu se pomoci rozhraní *Alpha* zredukuje aktuální teplota na teplotu novou a jako výstup metody je předán nejlepší nalezený jedinec.

5.11 Použití evolučních algoritmů

Pro použití těchto evolučních algoritmů slouží třída *Evolution*. Jejím úkolem je obalit použitý *EA* a poskytnout tak uživateli co nejjednodušší metody pro ovládání evoluce ve smyslu evolučního cyklu. Předkem třídy je třída *BaseEvolution*, která definuje základní metody pro práci s evolucí. Mezi ně patří především tyto abstraktní metody:

- **List<Individual> iteration()** – Provede jednu generaci (iteraci) evoluce a vrátí nově vzniklou populaci.
- **List<Individual> evolve()** – Provede evoluci a vrátí výsledek evoluce vzhledem k ukončovacím parametrům algoritmů.

Pro vytvoření instance třídy *Evolution* je potřeba specifikovat, jakým evolučním algoritmem bude evoluce probíhat, jaká je vstupní funkce, která se má fitovat, jaké funkce jsou v množině *GFS*, jak vypadá vzorový jedinec a jaké je nejhorší přípustné řešení pro zastavení evoluce.

Výpis kódu 19 ukazuje implementaci metody *evolve* ve třídě *Evolution*. Tato metoda reprezentuje evoluční cyklus. V první fázi se provede vynulování čítače počtu volání *CV* výpočtu kvůli statistikám. Následně se nastaví mez pro posílené hledání, resetuje se aktuální počet iterací a evoluční algoritmus se pomocí metody *reset* nastaví do inicializačního stavu. Evoluce probíhá v cyklu za podmínky, že aktuální evoluční algoritmus může dále provádět iteraci a zároveň ještě není nalezeno řešení požadované kvality. V každém takovém cyklu se provede volání implementace abstraktní metody *iteration*. Tato metoda vrátí novou populaci po provedení jedné generace evolučního algoritmu. Tento výsledek je následně uložen jako aktuální populace a je proveden přepočet *CV* pro každého jedince v populaci. Populace je poté seřazena od nejlepšího jedince po nejhoršího a na základě nejlepšího jedince je zavolána metoda *handleReinforcedSearch*, která v případě potřeby zapne posílené hledání a upraví meze možných indexů ve vzorovém jedinci pro příští cykly. Po skončení celého cyklu je evoluce dokončena a celá populace se seřadí podle nejlepšího jedince. Tento výsledek je pak návratovou hodnotou metody a představuje řešení optimalizovaného problému.

```
public List<Individual> evolve() {
    APCostValue.reset();
    double tr = −0.05;
    int actualIter = 0;
    mainAlg.reset();
    while (this.mainAlg.isNextIteration() && !hasGoodResult()) {
        List<Individual> newPop = iteration();
        results = newPop;
        updatePopulation(results);
        sortIndividuals(results);
        printIteration(actualIter++, results);
        fireIterationComplete(results);
        tr = handleReinforcedSearch(tr, results.get(0).getCostValue());
    }
    sortIndividuals(results);
    return results;
}
```

Výpis 19: Implementace evolučního cyklu

5.12 Evoluce konstant

Pokud požadujeme, aby se ve výrazu vyskytovaly i reálné konstanty, tak se evoluce takového výrazu komplikuje. Je totiž potřeba použít další evoluci, která se postará o jejich správný odhad. Pro evoluci takových výrazů je tedy potřeba použít evoluci v evoluci, kde jedna je nadřazená a druhá podřazená. Nadřazená evoluce má na starost složení výrazů z elementárních funkcí. Podřízená hledá nejlepší čísla hodící se pro konstanty výrazu, který je nadřazenou evolucí sestavován. Podřízená evoluce je tedy závislá na nadřízené.

CV jedince z podřízené evoluce lze totiž vypočítat pouze způsobem, že se tyto konstanty z tohoto jedince dosadí na místa konstant v jedinci nadřazené evoluce. Je to proto, že *CV* je závislé na tom, kde se jednotlivé konstanty ve výrazu nacházejí a jaké mají hodnoty. Tím se tedy evoluce celého výrazu časově komplikuje.

5.12.1 Jedinec

Pro evoluci konstant v programu existuje speciální typ jedince. Tato třída má název *IndividualConstantImpl*. Rozšiřuje třídu *IndividualImpl* o možnost uložení pole konstant, které má být použito pro substituci obecných typu konstant ve výrazu, který je z něho vytvořen. Jedinec takového typu je rozšířen o tyto metody:

- **double[] getConstants()** - Vráti pole konstant, které jsou v jedinci definovány.
- **int constantsSize()** - Vráti počet konstant, které jsou v jedinci definovány.
- **void setConstants(double[] constants)** - Nastaví pole konstant pro konkrétního jedince.
- **double getConstantAt(int index)** - Vrátí hodnotu určité konstany v jedinci na základě argumentu této metody.

Dále ze svého předka přepisuje metodu *update*, která je pozměněna tak, aby brala v potaz jeho rozšíření o konstanty. Tato metoda je upravena způsobem, který zobrazuje výpis kódu 20.

```
@Override
public void update() {
    Expression e = fi.createExpressionWithConstants(this);
    CostValue cv = new APCostValue(this, Source.getInstance());
    this.exp = e;
    this.costValue = cv.costValue();
}
```

Výpis 20: Upravená metoda update pro jedince s konstantama

Je zde tedy změna ve volání metody, která je zodpovědná za vytvoření instance třídy *Expression*. Volá se metoda *createExpressionWithConstants*, kde se jako parametr funkce předá jedinec, ze kterého je metoda volána. Tato metoda je až na jednu podmínku identická s metodou *createExpression*. Tím rozdílem je, že předtím, než je vracen výsledný výraz, je proveden test, zda je jedinec předaný v parametru funkce instancí *IndividualConstantImpl*. Pokud tomu tak je, tak jsou ve výrazu pomocí metody *injectConstants* nastaveny jednotlivé konstanty z jedince. Pokud tomu tak není, tak má tato metoda stejnou funkci jako metoda *createExpression*. Tuto změnu v kódu představuje kód z výpisu kódu 21.

```
if (individual instanceof IndividualConstantImpl) {
    exp.injectConstants(((IndividualConstantImpl) individual).getConstants(), 0);
}
```

Výpis 21: Rozdíl mezi metodou createExpressionWithConstants a createExpression

Jedinec předaný v parametru funkce je přetypován na typ *IndividualConstantImpl*. Následně je na něho zavolána metoda *getConstants*, která vrátí pole konstant z jedince. Toto pole je použito jako první parametr metody *injectConstants*. Druhý parametr této metody je pomocný, protože metoda pracuje rekurzivně. Metoda *injectConstants* prochází strom výrazu a hledá koncové uzly, které představují evoluční konstanty. Jejich hodnoty se následně snaží nastavit na hodnoty z pole předaného v prvním parametru funkce. Druhý parametr slouží pro uložení pozice ještě nepoužité konstanty z pole v prvním parametru.

5.12.2 Algoritmy

Každá implementace evolučního algoritmu se v programu vyskytuje ve dvou verzích. První slouží pro nadřízenou evoluci a implementuje rozhraní *EvolutionAlgorithm*. Druhá slouží pro podřízenou evoluci a implementuje rozhraní *ConstantEvolutionAlgorithm*. Rozdíl mezi těmito rozhraními je pouze v metodě *reset*. V rozhraní *ConstantEvolutionAlgorithm* má tato metoda dva parametry. První z nich definuje specimen objekt pro tvorbu konstant. Druhý říká, jaký jedinec bude pro instanci nadřazený. Díky této metodě není potřeba pro každého jedince z populace v nadřazené evoluci vytvářet vlastní instanci algoritmu. Stačí tak pouze jedna instance, kde nadřazený jedinec je definován metodou *reset*.

Jednotlivé algoritmy pro podřízenou evoluci v podstatě kopírují funkčnost jejich protějšků pro nadřazenou evoluci. Zásadní rozdíl zde však je v práci s jedincem. Algoritmy podřízené evoluce totiž pracují s jedinci odlišným způsobem. Všichni jedinci mají stejnou strukturu funkce a jediná věc, která je odlišuje, jsou hodnoty konstant. Při tvorbě nových jedinců je potřeba vycházet z nadřazeného jedince, ze kterého je v kombinaci s novým polem konstant vygenerovaným podřízenou evolucí vytvořen nový jedinec. Pro tyto operace obsahuje třída *Individual* statickou metodu *createIndividual*, která jako první parametr vyžaduje zdrojového jedince a jako druhý parametr pole konstant. Metoda následně vytvoří nového jedince, který má na základě poskytnutého pole konstant správně nastaveny svoje konstanty ve svém výrazu.

5.12.3 Iterace

Třída *Evolution* neobaluje pouze algoritmy pro nadřízenou evoluci, ale také pro podřízenou. Při vytváření instance této třídy lze v konstruktoru definovat, jaký algoritmus má být použit pro nadřízenou evoluci a jaký pro podřízenou. Z pohledu práce s evolučními algoritmy je ve třídě *Evolution* nejdůležitější metoda *iteration*. Ta má na starost správně použít algoritmy nadřízené i podřízené evoluce. Metoda *iteration* má implementaci zobrazenou ve výpisu kódu 22.

```
@Override
protected List<Individual> iteration () {
    List<Individual> mainResult = mainAlg.execute();
    int mainResultSize = mainResult.size();
    for (int individualIndex = 0; individualIndex < mainResultSize; individualIndex++) {
        Individual actual = mainResult.get(individualIndex);
        actual.update();
        int constCount = actual.getExpression().countOfConstantsToEvolve();
```

```
        if (constCount > 0) {
            Specimen constSpecimen = Specimen.createSpecimen(constCount,
                    -100, 100);
            constantAlg.reset(constSpecimen, actual);
            List<Individual> constIndividualsPopulation = new ArrayList<>();
            while (constantAlg.isNextIteration()) {
                List<Individual> newPop = constantAlg.execute();
                constIndividualsPopulation = newPop;
            }
            sortIndividuals(constIndividualsPopulation);
            IndividualConstantImpl bestConst = (IndividualConstantImpl) constIndividualsPopulation.
                get(0);
            mainResult.set(individualIndex, Individual.createIndividual(
                actual.getValues(), bestConst.getConstants(),
                actual.getMap()));
        }
    }
    return mainResult;
}
```

Výpis 22: Metoda pro iteraci evolučního cyklu

Nejprve je vytvořena nová generace jedinců pomocí volání metody *execute* na instanci algoritmu pro nadřazenou evoluci. Toto volání vrátí kolekci nových jedinců. Pro každého z nich se následně provede podřízená evoluce. Procházení těchto jedinců je realizováno cyklem *for*, kdy pro každého z nich se provedou následující aktivity. Nejprve se pro daného jedince vytvoří jeho reprezentace formou třídy *Expression*. Toho je docíleno voláním metody *update* na jedince. Následně se zjistí, kolik v sobě tento jedince obsahuje evolučních konstant. Toho je docíleno zavoláním metody *countOfConstantsToEvolve* na instanci jeho výrazu, který byl dříve pomocí metody *update* vytvořen. Pokud v sobě jedinec neobsahuje žádnou konstantu, tak cyklus pokračuje dalším jedincem. Naopak, pokud obsahuje alespoň jednu, je provedena podřízená evoluce, která má zjistit, jakých hodnot mají nabývat.

Činnost podřízené evoluce lze rozdělit do třech fází, které jsou provedeny po sobě. První fází je příprava algoritmu pro podřízenou evoluci. V této fázi je potřeba definovat specimen objekt pro tvorbu konstant. Je potřeba, aby obsahoval stejný počet prvků jako je konstant k evoluci. Tento specimen je spolu s aktuálním jedincem, pro kterého je tato podřízená evoluce prováděna, předán jako parametr metodě *reset* instanci algoritmu pro podřízenou evoluci. Tím algoritmus ví, pro jakého nadřazeného jedince bude evoluce probíhat a díky tomu může tvořit jedince, kteří mají stejnou strukturu jejich výrazu, ale jiné hodnoty konstant.

Další fází je samotná tvorba konstant pomocí evoluce. V té jsou vytvořeny konstanty, které budou později pro aktuálního jedince nastaveny. Tento proces probíhá v cyklu, dokud metoda *isNextIteration* z algoritmu pro podřízenou evoluci vrací logicky pravdivou hodnotu. V těle tohoto cyklu pak probíhá volání metody *execute* na evoluční algoritmus pro podřízenou evoluci, který vrátí kolekci jedinců (konstant). Tato kolekce je poté nastavena jako aktuální výsledek této fáze.

Poslední fází je nastavení těchto konstant aktuálnímu jedinci. Protože podřízená evoluce vytvoří celou populaci jedinců, kde jeden jedinec představuje pole konstant, je potřeba vybrat nejlepší řešení. Z toho důvodu je populace výsledných jedinců z podřízené populace seřazena podle *CV*, kde první prvek po seřazení představuje řešení podřízené evoluce. Z aktuálního jedince a takto získaného pole konstant se vytvoří nový jedinec, který nahradí aktuálního jedince ve výstupní kolekci.

5.13 Konfigurační soubor

Pro zjednodušení práce s programem, byl vytvořen konfigurační soubor. Díky němu uživatel programu nemusí při každé změně nastavení kód kompilovat. Konfigurace probíhá pomocí souboru ve formátu *JSON*, který se při spouštění programu z konzole předá jako vstupní parametr.

Samotný konfigurační soubor se skládá z pole *JSON* objektů, kde jejich počet a obsah musí splňovat předem stanovené kritéria. Hlavním konfiguračním objektem pro program je *JSON* objekt, který má klíč *global*. Jeho přítomnost a název klíče je povinná. Lze v něm nastavit možnosti, které jsou zobrazeny v tabulce 13, která se nachází v příloze práce.

Konfigurace jednotlivých evolučních algoritmů probíhá pomocí dalších *JSON* objektů, které jsou součástí konfiguračního souboru. V souboru jich může být libovolné množství, avšak pro jejich použití musí být jejich klíč uveden v atributu končícím na *_config* v globálním objektu. Samotné nastavení klíče však nestačí. Je ještě potřeba v atributu *algM* nebo *algS* zadat, o jaký typ algoritmu se jedná. Jejich možné nastavení je zobrazeno v tabulce 14, která se nachází v příloze práce. Význam všech atributů, které v rámci algoritmů lze nastavit se taktéž nachází v příloze v tabulce 15. Ukázka toho, jak takový konfigurační soubor může vypadat je zobrazena ve výpisu kódu 23.

```
{
    "global" : {
        "visualize" : true,
        "genomSize" : 30,
        "sampleFrequency" : 0.04,
        "sampleStart" : -1,
        "sampleEnd" : 1,
        "gfs" : "*+-/",
        "algM" : "SOMA_AllToAll",
        "algM_config" : "SOMA_AllToAll_Example",
        "algS" : "DE",
        "algS_config" : "DE_Example",
        "stopcost" : -0.0001,
        "f" : "pow(%x, 5) - 2 * pow(%x, 3) + pow(%x, 1)"
    },
    "DE_Example" : {
        "populationSize" : 10,
        "iterations" : 50,
        "CR" : 0.7,
        "f" : 0.02
    },
    "SOMA_AllToAll_Example" : {
        "populationSize" : 10,
```

```
        "iterations" : 10,
        "pathLength" : 1.5,
        "step" : 0.11,
        "prt" : 0.6
    }
}
```

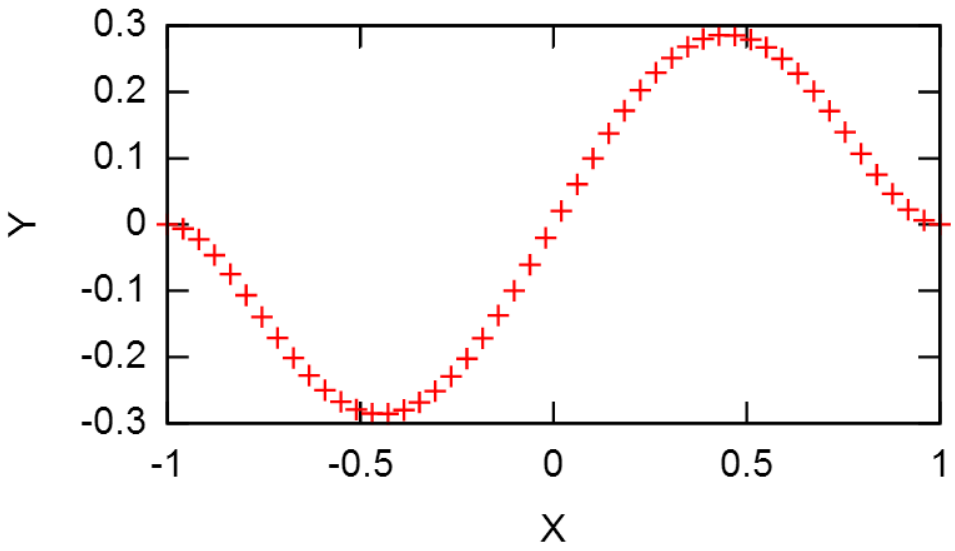
Výpis 23: Ukázka konfiguračního souboru pro program

6 Experimenty

Tato kapitola se zabývá experimentováním s implementovaným programem na vybraných problémech. Pro experimenty je zvoleno řešení problému prokládání vstupních dat funkcí, kde cílem je najít takovou funkci, která se bude co nejvíce podobat vstupní funkci, která je reprezentována jako množina bodů. Cílová funkce má předpis na základě vztahu 16. Tomuto problému se říká Quintic problém.

Pro všechny experimenty je samplování vstupní funkce provedeno po kroku 0.04 a to tím způsobem, že začíná v bodě -1 a končí v bodě 1. Tím vznikne celkem 50 bodů, které představují vstupní funkci. Tyto body jsou vyznačeny v obrázku 19. Velikost genomu jedince je nastavena na hodnotu 30. Pro všechny experimenty bylo povoleno posílené hledání.

$$y(x) = x^5 - 2x^3 + x \tag{16}$$



Obrázek 19: Funkce Quintic problému po samplování

6.1 Bez meta-evoluce

Experimenty v této kapitole se zabývají evolucí, kde součástí *GFS* nejsou evoluční konstanty. Jsou zde otestovány celkem dva algoritmy, u kterých je nejprve ukázáno, jak vypadá průběh jednoho jejich běhu evoluce a následně jak se chovají statisticky, pokud je těchto běhů provedeno mnoho.

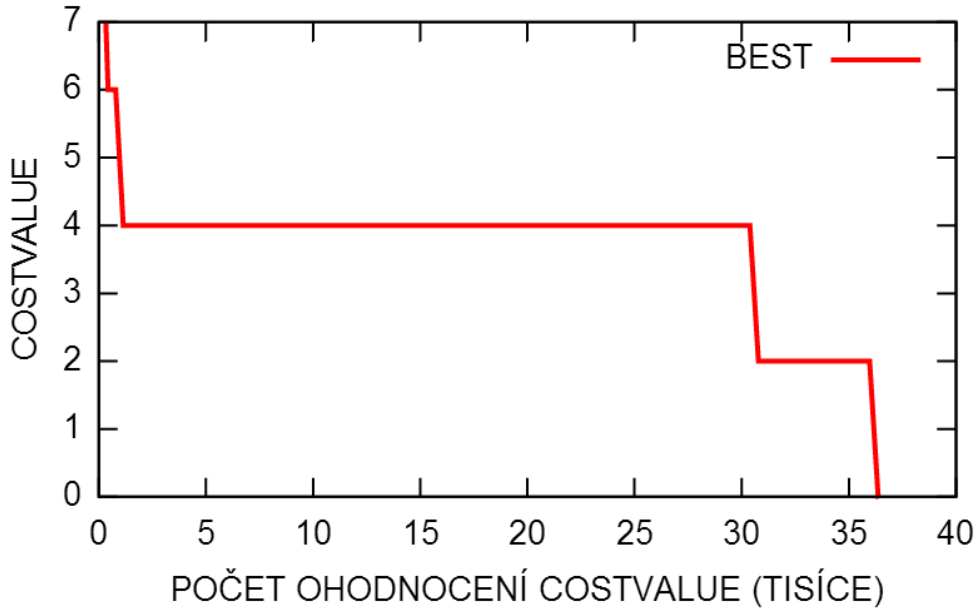
6.1.1 Diferenciální evoluce

První experiment se týká algoritmu *DE*, kde jednotlivé parametry prostředí jsou nastaveny způsobem, který zobrazuje tabulka 9.

Velikost populace	100
Počet generací	100
CR	0.8
F	0.3
GFS	+-*/

Tabulka 9: Nastavení algoritmu DE pro experiment

Na obrázku 20 je zobrazen výsledek jednoho běhu evoluce. Jedná se o závislost *CV* na počtu ohodnocení jedince pomocí účelové funkce. Červená křivka reprezentuje vždy hodnotu *CV*, kterou nabýval nejkvalitnější jedinec v populaci při určitém počtu ohodnocení *CV*.

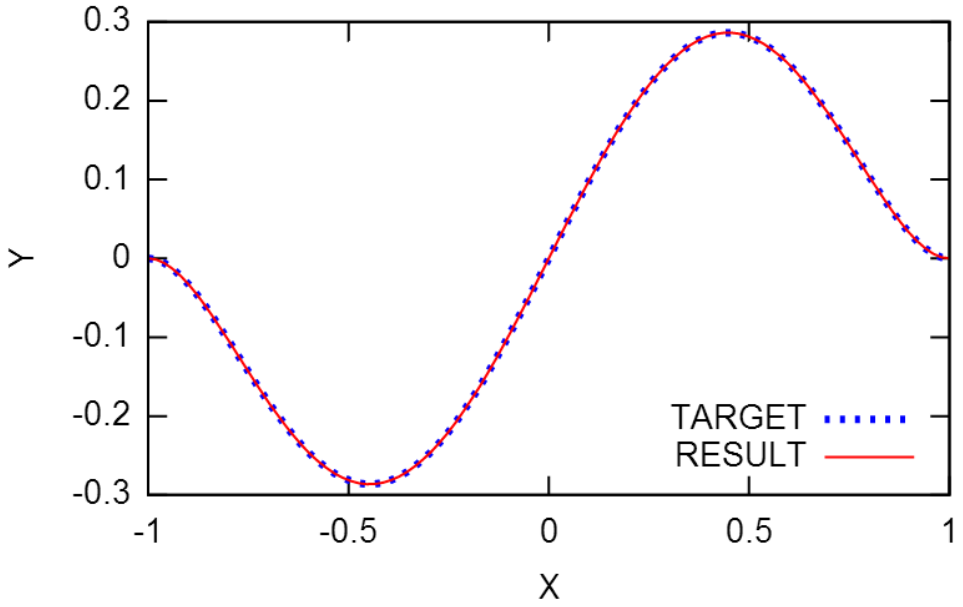


Obrázek 20: Vývoj řešení jednoho běhu algoritmu DE

Výsledkem tohoto experimentu je předpis funkce, který je před matematickou úpravou reprezentován výrazem 17 a po úpravě výrazem 18. Graf této funkce představuje obrázek 21, kde lze vidět, že byla nalezena adekvátní funkce, která vstupní funkci velice věrně aproximuje.

$$y(x) = ((((((((((x - x)\,x + x) + x) + x) - x) - x)\,x)\,x - x) - x)\,x) + x \tag{17}$$

$$y(x) = \left(x^3 - 2x\right) x^2 + x \tag{18}$$



Obrázek 21: Graf nalezené fukce algoritmem DE

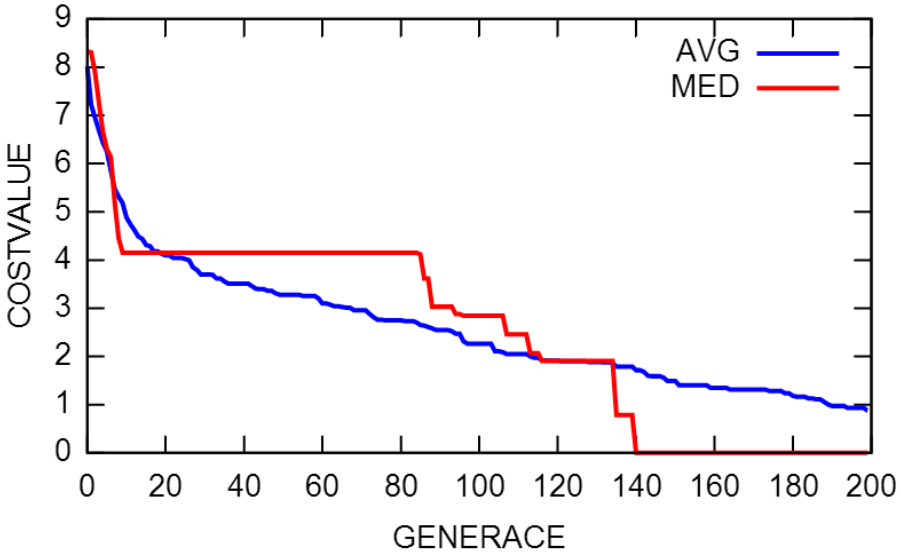
Na obrázku 22 je zobrazeno, jak vypadá situace, kdy je evoluce pro *DE* s počtem iterací 200 spuštěna $80x$. Obrázek představuje závislost *CV* na počtu generací. Modrá křivka představuje průměrnou kvalitu řešení přes všech 80 běhů evoluce v závislosti na počtu dokončených generací. Červená křivka reprezentuje medián přes tyto všechny běhy v závislosti na počtu dokončených generací. Z grafu lze vyčíst, že průměrná kvalita řešení se v závislosti na generacích zvětšuje.

V rámci experimentu byly syntetizovány například následující funkce:

$$y(x) = x - \left(\left(\left(\left(\left(x\left((x * x) * \frac{x - x}{x} - x\right)\right) x + x\right) + x\right) - x\right) + x\right) x \tag{19}$$

$$y(x) = \left(\left(\left(\left(\left(\left(\left(\left(\frac{x - x}{x} + x\right) - x\right) + x\right) x\right) x - x\right) - x\right) + x\right) - x\right) x\right) x + x \tag{20}$$

$$y(x) = \left(\left(\left(\left(\left(x - \left(\left(\frac{\frac{x-x}{x}}{x} x - x\right) x\right) x\right) - x\right) - x\right) - x\right) x\right) x + x \tag{21}$$



Obrázek 22: Statistické chování algoritmu DE

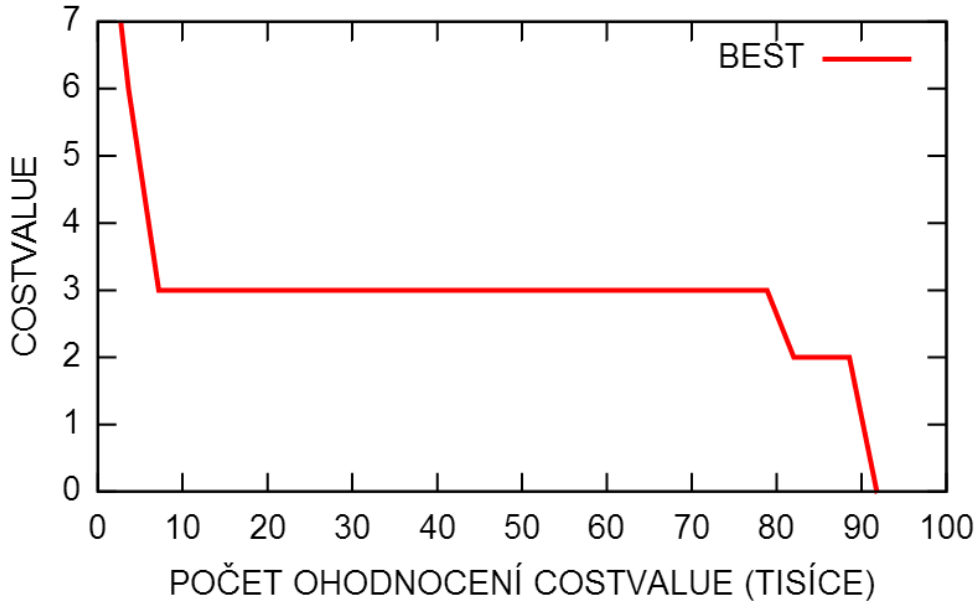
6.1.2 SOMA

Tento experiment se týká algoritmu *SOMA* se strategií „Všichni k jednomu“, kde jednotlivé parametry prostředí jsou nastaveny způsobem, který zobrazuje tabulka 10.

Velikost populace	30
Počet migrací	100
Step	0.11
PathLength	1.5
PRT	0.6
GFS	+-*/

Tabulka 10: Nastavení algoritmu SOMA pro experiment

Na obrázku 23 je zobrazen výsledek jednoho běhu evoluce. Jedná se o závislost *CV* na počtu ohodnocení jedince pomocí účelové funkce. Červená křivka reprezentuje vždy hodnotu *CV*, kterou nabýval nejkvalitnější jedinec v populaci při určitém počtu ohodnocení *CV*.

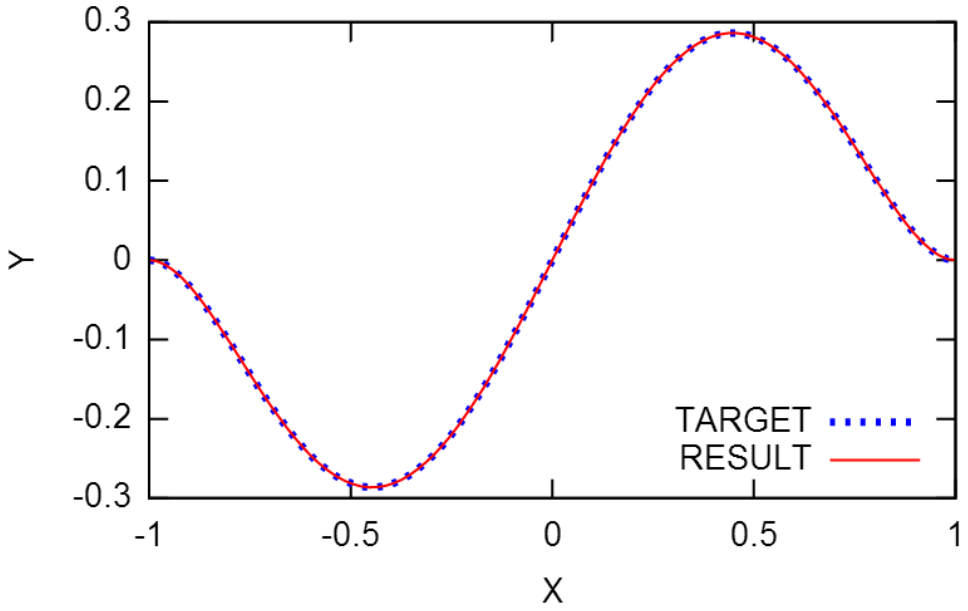


Obrázek 23: Vývoj řešení jednoho běhu algoritmu SOMA

Výsledkem tohoto experimentu je předpis funkce, který je před matematickou úpravou reprezentován výrazem 22 a po úpravě výrazem 23. Graf této funkce představuje obrázek 24, kde lze vidět, že byla nalezena v podstatě totožná funkce, jak hledaná funkce, však s jiným předpisem.

$$y(x) = x - ((((((((((x - x)\,x - x)\,x + x) - x)\,x + x) - x) + x) + x)\,x)\,x \tag{22}$$

$$y(x) = x - x^2\left(2x - x^3\right) \tag{23}$$



Obrázek 24: Graf nalezené fukce algoritmem SOMA

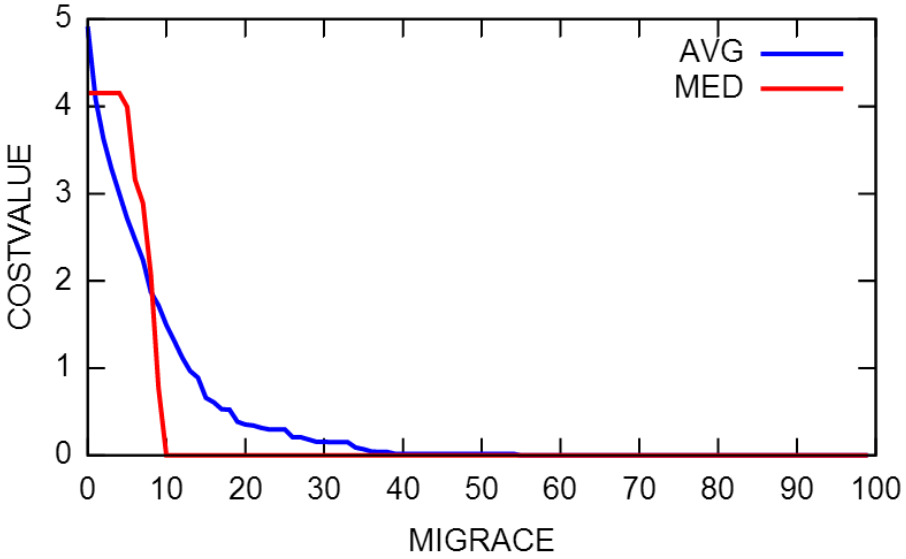
Na obrázku 25 je zobrazeno, jak vypadá situace, kdy je evoluce pro algoritmus *SOMA* spuštěna $80x$. Obrázek představuje závislost *CV* na počtu migrací. Modrá křivka představuje průměrnou *CV* přes všech 80 běhů evoluce v závislosti na počtu dokončených migrací. Červená křivka reprezentuje medián přes tyto všechny běhy v závislosti na počtu dokončených migrací. Z grafu lze vyčíst, že průměrná kvalita řešení se v závislosti na migracích zvětšuje. Algoritmus velice rychle konverguje k hledanému řešení. Již při desáté migraci polovina jedinců z populace nalezla odpovídající funkci.

V rámci experimentu byly syntetizovány například následující funkce:

$$y(x) = x - ((((((((((x - x) + x) - x)\,x - x)\,x)\,x - x) + x) + x) + x)\,x)\,x \tag{24}$$

$$y(x) = \left(\left(\left(\left(\left(x + \frac{x - (x + (x - x)\,x)}{x}x\right)x\right)x - x\right) - x\right)x\right)x + x \tag{25}$$

$$y(x) = (((((((x\,((x - x)\,x) + x) - x)\,x + x)\,x)\,x - x) - x)\,x)\,x + x \tag{26}$$



Obrázek 25: Statistické chování algoritmu SOMA

6.2 S meta-evolucí

Experimenty v této kapitole se zabývají evolucí, kde součástí *GFS* jsou evoluční konstanty, tedy pro každého jedince je proveden odhad jeho konstant pomocí podřízeného evolučního algoritmu. Je zde otestována situace, kdy nadřazený algoritmus je *SOMA* a podřízený *PS*. Je zde nejprve ukázáno, jak se vyvíjí kvalita výsledného jedince v závislosti na počtu ohodnocení účelové funkce a následně jak se tato kombinace algoritmů chová statisticky.

6.2.1 SOMA + PS

Tento experiment se týká algoritmu *SOMA* se strategií „Všichni k jednomu“ včetně meta-evoluce, kde pro odhad konstant je použit algoritmus *PS*. Parametry algoritmu *SOMA* jsou nastaveny na základě tabulky 11. Parametry algoritmu *PS* jsou nastaveny na základě tabulky 12. Množina *GFS* obsahuje funkce „+-*/C“.

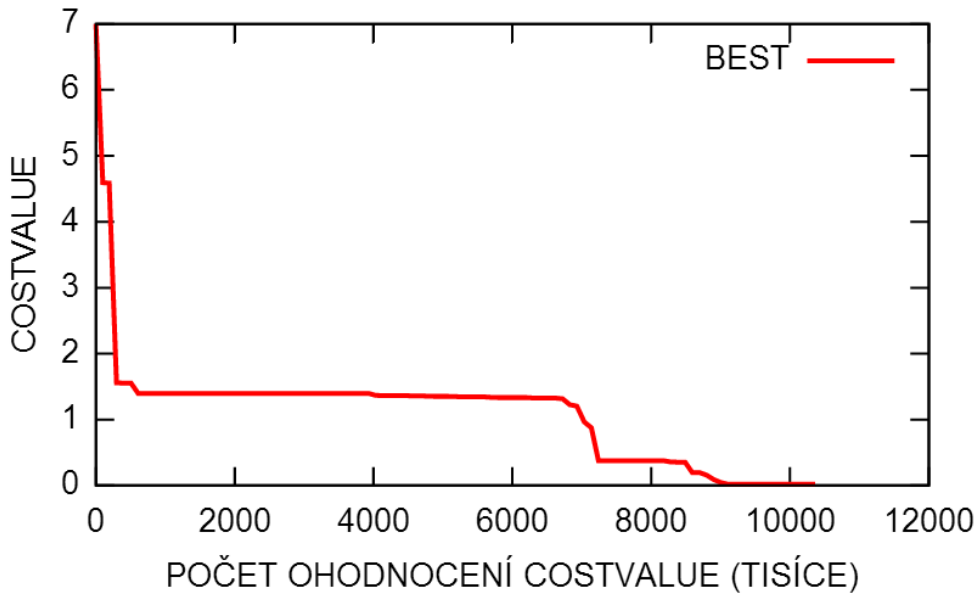
Velikost populace	20
Počet migrací	100
Step	0.11
PathLength	1.5
PRT	0.6

Tabulka 11: Nastavení algoritmu SOMA pro experiment s meta-evolucí

Velikost populace	50
Počet generací	100
c1	2
c2	2
wStart	0.9
wEnd	0.4
vmax	2.5

Tabulka 12: Nastavení algoritmu PS pro experiment s meta-evolucí

Na obrázku 26 je zobrazen výsledek jednoho běhu meta-evoluce. Jedná se o závislost *CV* na počtu ohodnocení jedince pomocí účelové funkce. Červená křivka reprezentuje vždy hodnotu *CV*, kterou nabýval nejkvalitnější jedinec v populaci při určitém počtu ohodnocení *CV*. Výsledkem tohoto experimentu je předpis funkce, který se z důvodu své velikost nachází v příloze (B).



Obrázek 26: Vývoj řešení jednoho běhu meta-evoluce

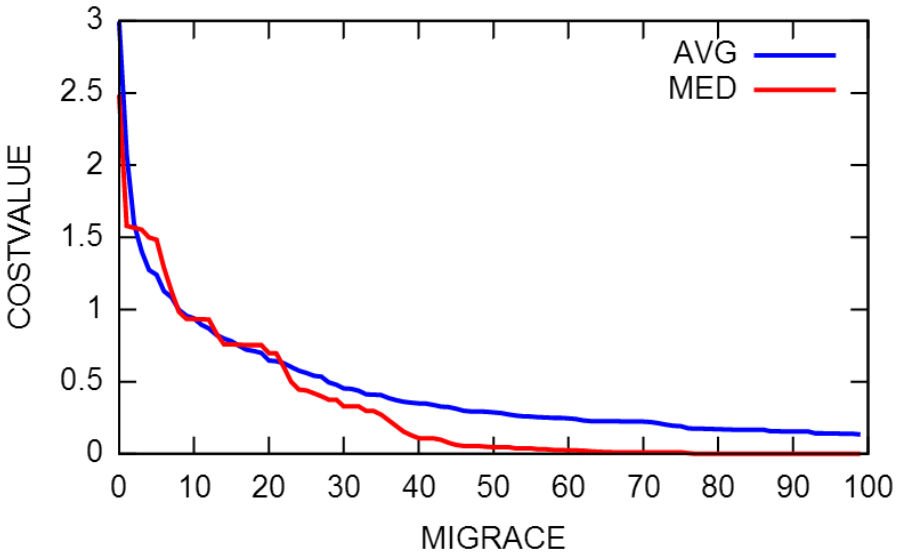
Na obrázku 27 je zobrazeno, jak vypadá situace, kdy je meta-evoluce pro algoritmus *SOMA* jako nadřazený a *PS* jako podřazený spuštěna $80x$. Obrázek představuje závislost *CV* na počtu migrací. Modrá křivka představuje průměrnou *CV* přes všech 80 běhů evoluce v závislosti na počtu dokončených migrací. Červená křivka reprezentuje medián přes tyto všechny běhy v závislosti na počtu dokončených migrací. Z grafu lze vyčíst, že meta-evoluce již při padesáté migraci má více jak polovinu populace, která obsahuje kvalitní řešení.

V rámci experimentu byly syntetizovány například následující funkce:

$$y(x) = \frac{-0.00133}{83.52738} - (((((x - ((3.56909x + x)\,x - 0.9994))\,0.99963)\,x + x)\,x)\,x - x) \quad (27)$$

$$y(x) = \left(\left(\left(x\left(\left(\left(\frac{(86.52141 - (-85.50371)\,x) + x}{86.53956} - x\right)x\right)x\right) - x\right) - x\right)x\right)x + x \quad (28)$$

$$y(x) = \left(1 - \left(\left(\left(\left(\left(\left((x - x) * \frac{78.55408 - x}{x} - x\right)x\right)x + x\right) + x\right) + x\right) - x\right)x\right)x \quad (29)$$



Obrázek 27: Statistické chování meta-evoluce při použití algoritmu SOMA jako nadřízeného algoritmu a PS jako podřízeného

7 Závěr

Tato diplomová práce se zabývala implementací metody *AP*. V teoretické části práce byl vysvětlen princip této metody včetně jednotlivých evolučních algoritmů, které byly při implementaci použity. V implementační části byla popsána a vysvětlena tvorba programu včetně okomentovaných zdrojových kódů vybraných částí programu. Tento popis programu slouží zároveň jako jeho dokumentace. Byly popsány také jednotlivé rozhraní tříd použitých v rámci implementace.

Program pro *AP* byl vytvořen v jazyce Java jako aplikace, která se ovládá pomocí konzole. Celá implementace byla rozdělena do dvou komponent. První z nich představuje samotnou konzolovou aplikaci, která má na starost pracovat s druhou komponentou. Ta představuje samotné jádro *AP*. V programu bylo implementováno celkem 10 různých algoritmů, které si může uživatel pro evoluci zvolit. V rámci množiny *GFS* bylo implementováno 12 základních funkcí. Program umí provádět také meta-evoluce pro odhady konstant určených pro evoluci. Uživatelské nastavení evoluce lze provést pomocí konfiguračního souboru. Program byl implementován s ohledem na jeho budoucí rozšiřitelnost a tak nové funkce a algoritmy do něho lze lehce doimplementovat.

Pro ověření správného fungování programu bylo provedeno za použití různých evolučních algoritmů několik testů. Řešeným testovaným problémem byla aproximace funkce, kde cílem programu je najít předpis takové funkce, která co nejlépe protíná dané body. Koncepčně se jednalo o testy jak s meta-evolucí, tak bez meta-evoluce. Ukázalo se, že výsledky z těchto experimentů lze považovat za dostatečné, protože byly nalezeny předpisy funkcí požadované kvality. To dokazují také grafy, které byly na základě těchto experimentů vytvořeny. Lze tedy říci, že implementace metody *AP* proběhla úspěšné.

Výsledný program jako takový lze využít například pro experimentování s *AP* nebo pro vyuku fungování této techniky. Samozdřejmě ho lze také použít na řešení uživatelem definovaných problémů. Program také lze dále rozšiřovat. Možností je hned několik, například pro něho vytvořit ovládání pomocí *GUI*, vytvořit modul pro zautomatizovaní experimentování nebo doimplementovat další algoritmy.

Bc. Tomáš Tyleček

8 Reference

[1] *On the Origin of Circuits* [online]. [cit. 12. 4. 2014]. Dostupné z: <http://www.damninteresting.com/on-the-origin-of-circuits/>.

[2] *TIOBE Index for April 2014* [online]. [cit. 12. 4. 2014]. Dostupné z: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.

[3] KVASNIčKA V., T. P. P. J. *Evolučné algoritmy*. Bratislava : STU Bratislava, 2000c. ISBN 85-246-2000.

[4] ZELINKA, I. *Evoluční výpočetní techniky: principy a aplikace*. BEN-technická literatura, 2009d.

[5] FOGEL, G. B. – CORNE, D. W. *Evolutionary computation in bioinformatics*. Morgan Kaufmann, 2002e.

[6] ZELINKA, I. et al. *Evolutionary algorithms and chaotic systems*. 267. Springer, 2010f.

[7] *Evolutionary Design of Antennas* [online]. [cit. 12. 4. 2014]. Dostupné z: <http://idesign.ucsc.edu/projects/evo_antenna.html>.

[8] J.R., K. *Genetic Programming*. MIT Press, 1998h. ISBN 0-262-11189-6.

[9] KOZA J.R., A. D. K. M. B. F. *Genetic Programming III*. Morgan Kaufnamm pub., 1999i. ISBN 1-55860-543-6.

[10] O'NEILL, M. – RYAN, C. *Grammatical evolution: evolutionary automatic programming in an arbitrary language*. 4. Springer, 2003j.

[11] ZELINKA, I. – OPLATKOVA, Z. – NOLLE, L. Analytic programming-symbolic regression by means of arbitrary evolutionary algorithms. *Int. J. of Simulation, Systems, Science and Technology*. 2005k, 6, 9, s. 44–56.

[12] ZELINKA, I. et al. Analytical programming-a novel approach for evolutionary synthesis of symbolic structures. *Evolutionary Algorithms. InTech*. 2011l.

[13] AUGER, A. Convergence results for the (1, λ)-SA-ES using the theory of ϕ-irreducible Markov chains. *Theoretical Computer Science*. 2005m, 334, 1, s. 35–69.

[14] COELLO, C. A. C. – AGUIRRE, A. H. – ZITZLER, E. *Evolutionary multi-criterion optimization*. Springer, 2005n.

[15] BÄCK, T. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996o.

[16] PRICE, K. – STORN, R. M. – LAMPINEN, J. A. *Differential evolution: a practical approach to global optimization*. Springer, 2006p.

[17] LAMPINEN JOUNI, Z. I. *New Ideas in Optimization and Mechanical Engineering Design Optimization by Differential Evolution*. Volume 1. London : McGraw-Hill, 1999q. ISBN 0-07-709506-5.

[18] STORN, R. On the usage of differential evolution for function optimization. In *Fuzzy Information Processing Society, 1996. NAFIPS., 1996 Biennial Conference of the North American*, s. 519–523. IEEE, 1996r.

[19] CHAKRABORTY, U. K. et al. *Advances in differential evolution*. Springer, 2008s.

[20] ZELINKA, I. SOMA—self-organizing migrating algorithm. In *New optimization techniques in engineering*. Springer, 2004t. s. 167–217.

[21] CLERC, M. *Particle swarm optimization*. 93. John Wiley & Sons, 2010u.

[22] KENNEDY, J. – EBERHART, R. et al. Particle swarm optimization. In *Proceedings of IEEE international conference on neural networks*, 4, s. 1942–1948. Perth, Australia, 1995v.

[23] *Simulované žíhání* [online]. [cit. 14. 4. 2014]. Dostupné z: <http://www.fit.vutbr.cz/~jarosjir/groups/eva/sa.html.cs>.

[24] VAN LAARHOVEN, P. J. – AARTS, E. H. *Simulated annealing*. Springer, 1987x.

[25] *The Java Language Environment* [online]. [cit. 14. 4. 2014]. Dostupné z: <http://www.oracle.com/technetwork/java/intro-141325.html>.

[26] *TIOBE Index for April 2014* [online]. [cit. 14. 4. 2014]. Dostupné z: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.

[27] *Programovací jazyk Java* [online]. [cit. 14. 4. 2014]. Dostupné z: <http://programujte.com/clanek/2007040702-java-tutorial-technologie-1-dil/>.

[28] *Objekty - Singleton Pattern* [online]. [cit. 17. 4. 2014]. Dostupné z: <http://objekty.vse.cz/Objekty/Vzory-Singleton>.

[29] *Dealing with Properties* [online]. [cit. 12. 4. 2014]. Dostupné z: <http://martinfowler.com/apsupp/properties.pdf>.

A Možnosti konfigurace programu

Klíč	Popis klíče	Možná hodnota
visualize	Má se výsledek vizualizovat?	true, false
genomSize	Maximální velikost jedince	Celé nezáporné číslo
sampleFrequency	Krok vzorkování	Číslo
sampleStart	Počáteční bod vzorkování	Číslo
sampleEnd	Konečný bod vzorkování	Číslo
gfs	Seznam funkcí, které se nacházejí v GFS	* : násobení
		+ : sčítání
		- : odečítání
		/ : dělení
		s : sinus
		c : cosinus
		t : tangens
		n : negace
		a : absolutní hodnota
		C : konstanta
stopcost	Požadovaná kvalita CV, při které se evoluce zastaví	Číslo
f	Vstupní funkce	Předpis vstupní funkce, kde proměnná je zapsána jako „%x“
		např: ”pow(%x, 5)”
algM	Jaký evoluční algoritmus má být použit pro evoluci	DE
		SOMA_AllToOne
		SOMA_AllToOneRand
		SOMA_AllToAllAdaptive
		SOMA_AllToAll
		ES_OnePlusOne
		ES_Rekombinant
		ES_Multi
		PS
		SA
algM_config	Název JSON objektu pro konfiguraci algoritmu pro evoluci	např: ”DE_mujconfig”
algS	Jaký evoluční algoritmus má být použit pro evoluci konstant	DE
		SOMA_AllToOne
		SOMA_AllToOneRand
		SOMA_AllToAllAdaptive
		SOMA_AllToAll
		ES_OnePlusOne
		ES_Rekombinant
		ES_Multi
		SA
		PS
		SA
algS_config	Název JSON objektu pro konfiguraci algoritmu pro evoluci konstant	např: ”SOMA_mujconfig”

Tabulka 13: Možnosti konfigurace globálního nastavení

Zkratka	Výnam	Atribut
DE	Diferenciální evoluce	populationSize
		iterations
		CR
		f
SOMA_AllToOne, SOMA_AllToOneRand, SOMA_AllToAll, SOMA_AllToAllAdaptive	všichni k jednomu, všichni k jednomu náhodně, všichni ke všem, adaptivně všichni ke všem	populationSize
		iterations
		pathLength
		step
		prt
ES_OnePlusOne	Evoluční strategie (dvoučlenná)	iterations
		standartDeviation
		CV
ES_Rekombinant	Evoluční strategie (rekombinační)	populationSize
		iterations
		standartDeviation
		CV
		recombCount
		isDiscrete
ES_Multi	Evoluční strategie (vícečlenná)	populationSize
		iterations
		standartDeviation
		CV
SA	Simulované žíhání	tStart
		tEnd
		metro_iter
		neighbourSize
PS	Particle swarm	populationSize
		iterations
		c1
		c2
		wStart
		wEnd
		vmax

Tabulka 14: Možnosti konfigurace jednotlivých algoritmů

Atribut	**Popis atributu**
populationSize	Velikost populace
iterations	Počet iterací
CR	Práh křížení
f	Mutační konstanta
pathLength	Délka cesty
step	Krok
prt	Perturbace
standartDeviation	Směrodatná odchylka
CV	Minimální požadovaná CV pro ukončení algoritmu
recombCount	Počet rekombinací
isDiscrete	Jestli je rekombinace diskrétní nebo ne
tStart	Počáteční teplota
tEnd	Konečná teplota
metro_iter	Počet opakování Metropolisova algoritmu
neighbourSize	Počet sousedů
c1	Učící faktor
c2	Učící faktor
wStart	Počáteční setrvačnost
wEnd	Konečná setrvačnost
vmax	Maximální rychlost částic

Tabulka 15: Význam jednotlivých atributů pro konfiguraci algoritmů

B Výsledná funkce získána meta-evolucí

$y(x) = (x + ((((((((((((x/(x)) - 12,15802) - x) + x) + 12,1559) * x) * x) - x) + x) - x) - x) * x) * x))$